Практика программирования

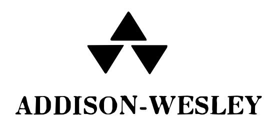
Брайан Керниган
Роб Пайк



Серия "Программирование для профессионалов" от издательства Addison-Wesley

The Practice of Programming

Brian W. Kernighan
Rob Pike



ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Практика программирования

Брайан Керниган
Роб Пайк



Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2015

ББК 32.973.26-018.2.75
К36
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.Л. Бродового*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Керниган, Брайан У., **Пайк**, Роб.

К36 Практика программирования. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 288 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-2005-8 (рус.)

Вашему вниманию предлагается перевод на русский язык исправленного и дополненного издания (уже восьмого по счету) популярной книги, вышедшего из печати в январе 2004 года. Верификацию кода в русском издании выполнили сами авторы книги — Брайан Керниган и Роб Пайк, что лишний раз свидетельствует об их огромной ответственности перед читателями.

В книге рассматриваются принципы практического профессионального программирования, которые, выходя за рамки простого написания кода, включают в себя проектирование, правильный выбор алгоритмов и структур данных, отладку и тестирование, оптимизацию быстродействия и переносимости, автоматизацию рабочего процесса. Изложение проиллюстрировано примерами из сложных, практически важных систем.

Книга предназначена для повышения квалификации программистов. Может быть полезна студентам и преподавателям компьютерных специальностей.

ББК 32.973.26-018.2.75

ISBN 978-5-8459-2005-8 (рус.)
ISBN 0-201-61586-X (англ.)

Оглавление

Содержание

Предисловие

Приходилось ли вам когда-нибудь

тратить уйму времени на программирование неправильного алгоритма?
пользоваться структурой данных, сложность которой превосходила всякое понимание?
пропускать ошибку, лежащую на поверхности, даже при самом тщательном тестировании программы?
возиться целый день с исправлением ошибки, на которую следовало бы потратить пять минут?
дорабатывать программу так, чтобы она работала втрое быстрее и при этом потребляла меньше памяти?
в муках переносить программу с рабочей станции на персональный компьютер или наоборот?
вносить “косметические” изменения в программы, написанные кем-то другим?
переписывать программу начисто, потому что она была совершенно неудобочитаема?

Скорее всего, вам это не понравилось. Что ж, подобное случается с программистами постоянно. Решение этих проблем часто оказывается более трудоемким, чем следовало бы, потому что такие темы, как тестирование, отладка, переносимость, быстродействие, стиль, проектирование программ — т.е. *практика программирования*, — вовсе не являются центральными в вузовских курсах программирования и вычислительной техники. Большинству программистов приходится осваивать все это хаотически, по мере накопления опыта, а некоторым из них вообще не судьба этому научиться.

В мире объемных и изощренных интерфейсов, постоянно изменяющихся языков, сред и утилит разработки, под давлением необходимости осваивать все новые и новые знания нетрудно потерять основной ориентир — такие базовые принципы, как простоту, ясность, общность, которые образуют фундамент для разработки высококачественных программ. Увы, людям свойственно переоценивать роль стандартов и автоматизированных сред программирования, которые механизируют значительную часть разработки программ, так что компьютер в известной степени становится сам себе программистом.

Подход, принятый нами в этой книге, основан именно на глубоких и взаимосвязанных принципах, применимых в программировании любого уровня. Эти принципы следующие. *Простота* означает, что программа должна быть короткой и легкой для доработки; *ясность* состоит в том, что ни у программиста при чтении програм-

мы, ни у компьютера при ее выполнении не должно возникать сомнений и ощущения двусмысленности; *общность* заключается в способности программы правильно реагировать на широкий диапазон ситуаций и легко адаптироваться к новой обстановке; наконец, *автоматизация* (ради которой, собственно, и пишутся программы) призвана освободить человека от трудоемких и однообразных операций. Рассматривая искусство программирования на самых разных языках и в широкой предметной области — от алгоритмов и структур данных до архитектуры, отладки, тестирования и повышения быстродействия, — можно проиллюстрировать универсальные инженерные концепции, независимые от конкретного языка, операционной системы или среды программирования.

Эта книга возникла как результат многолетнего опыта в области разработки и доработки самого разного программного обеспечения, преподавания курсов программирования и вычислительной техники, а также работы с огромным количеством программистов. Нам хочется поделиться усвоенными практическими уроками, предложить целый ряд методов и техник, которые бы помогли программистам всех уровней повысить свою квалификацию и производительность труда.

Мы пишем для нескольких категорий читателей. Если вы — студент, уже прошли несколько предметов из курса программирования и вычислительной техники и хотите лучше научиться программировать практические задачи, то в этой книге вы найдете более подробное изложение некоторых тем, недостаточно освещенных в учебной программе. Если вы пишете программы по роду своей деятельности, но скорее в качестве вспомогательных средств, а не основных продуктов производства, то приведенные в книге сведения помогут вам программировать на этом уровне более эффективно. Предлагаемый материал окажется полезным также в том случае, если вы — профессиональный программист, но в свое время не получили достаточных знаний по тем или иным вопросам из программы вуза (или же хотели бы освежить их в памяти). Наконец, немало интересного найдет здесь руководитель коллектива по разработке программного обеспечения, желающий научить своих подчиненных работать лучше.

Надеемся, что изучение материала этой книги пойдет на пользу вашим программам. Единственное требование к подготовке читателя — это некоторый опыт программирования, лучше всего на языках C, C++ или Java. Конечно, чем больше опыта, тем лучше; никто и ничто не может сделать специалиста из новичка за две-три недели. Программисты для систем Unix и Linux найдут в наших примерах больше знакомых мест и характерных черт, чем те, кто работал только в средах Windows и Macintosh. Тем не менее узнать что-то новое из книги и тем самым облегчить свою работу сможет практически каждый читатель, в какой бы среде он ни программировал.

Материал книги подразделяется на девять глав, каждая из которых посвящена одному большому и важному аспекту практического программирования.

В главе 1 рассматривается стиль программирования. Хороший стиль настолько важен для успешной работы программиста, что мы решили поставить его на первое место в книге. Хорошо написанные программы работают лучше, чем написанные плохо, — в них меньше ошибок, и их легче исправлять или отлаживать. Поэтому о хорошем стиле нужно помнить с самого начала. В этой же главе сделано введение в

такую важную тему, как использование устойчивых конструкций, характерных для того или иного языка.

Глава 2 посвящена алгоритмам и структурам данных, которые традиционно составляют основу вузовских учебных планов по программированию и вычислительной технике. Поскольку большинство читателей наверняка знакомы с этой тематикой, в нашем изложении мы дадим лишь краткий обзор основных алгоритмов и структур данных, используемых практически во всех программах. Более сложные конструкции обычно строятся их этих элементарных "кирпичиков", поэтому их знание имеет основополагающий характер.

В главе 3 рассматриваются архитектура и реализация небольшой программы, которая иллюстрирует вопросы алгоритмизации и структуризации данных в реалистической манере, приближенной к практической работе. Эта программа написана на пяти языках. Сравнение различных ее версий позволяет оценить, как одни и те же структуры данных реализованы в каждом из языков и как отличаются выразительность и эффективность аналогичных конструкций в различных средах программирования.

Способы взаимодействия (интерфейсы) пользователя, программ и отдельных частей программ составляют фундамент программирования, так что успех той или иной программы в значительной мере определяется тем, насколько хорошо спроектированы и реализованы ее интерфейсы. В главе 4 рассказывается о том, как эволюционировала небольшая библиотека для синтаксического анализа популярного формата данных. Хотя пример и невелик, в нем иллюстрируются многие важные вопросы проектирования программ: абстрагирование, сокрытие данных, управление ресурсами, обработка ошибок.

Как бы мы ни старались написать программу безошибочно с первой же попытки, все же возникновение ошибок в ней неизбежно. В главе 5 излагаются общая стратегия и тактические приемы, помогающие отлаживать программы и исправлять ошибки более систематически и эффективно. Среди рассмотренных в этой главе тем — формы проявления типичных ошибок и анализ статистики, позволяющий выявить источник проблемы по некоторым типичным шаблонам в отладочных выходных данных.

Тестирование представляет собой попытку дать достаточное доказательство того, что программа работает правильно и остается правильной по мере ее доработки. Поэтому в главе 6 мы сосредоточимся на вопросах тестирования — как выполняемого вручную, так и автоматизированного. Например, потенциально опасные или слабые места в программах можно обнаружить проверкой граничных условий. С помощью различных средств автоматизации (наподобие специально написанных для этого программ или сценариев) можно легко выполнить большие объемы тестов сравнительно небольшими усилиями. Совершенно другая типичная разновидность тестирования, предназначенная для выявления других видов ошибок, называется *стрессовым тестированием* и также рассматривается в этой главе.

Компьютеры стали такими быстрыми, а компиляторы — настолько качественными, что многие программы вполне удовлетворяют требованиям к их быстродействию с первого же дня их работы. Но бывают и программы, работающие слишком медленно или потребляющие слишком много памяти, а некоторые из них отличаются сразу обоими этими недостатками. В главе 7 представлен систематический подход к эффективной организации ресурсов, используемых программой, который позволяет

сохранить правильность и рациональность организации программы в ходе оптимизации ее работы.

В главе 8 рассматриваются вопросы переносимости. Действительно успешные программы существуют достаточно долго, так что их выполняющая среда успевает неоднократно измениться. Может также возникнуть необходимость переноса таких программ на новую аппаратную платформу, в новую операционную или языковую среду. Хорошая переносимость программы в этом контексте означает минимум труда и времени, которые необходимо затратить на ее приспособление к новой среде.

В программировании существует множество языков, причем не только языков общего назначения, используемых для реализации разнообразных универсальных задач, но и узкоспециализированных, предназначенных для ограниченной области приложений. В главе 9 демонстрируется важность знаковых систем или систем обозначений, которые можно использовать для упрощения программ, структуризации стоящих перед программистом задач и даже для написания программ, которые сами способны генерировать другие программы.

Говоря о программировании, поневоле приходится демонстрировать большие объемы кода. Большинство примеров кода было написано специально для этой книги, и лишь некоторые небольшие фрагменты заимствованы из других источников. Мы приложили все усилия к тому, чтобы этот код был качественным, и протестировали его в пяти-шести системных средах непосредственно в электронной форме. Более подробную информацию об этом можно найти на Web-сайте, посвященном данной книге:

```
http://tpop.awl.com
```

Большая часть приведенных примеров написана на языке C; имеется некоторое количество кода на C++ и Java, а также совсем небольшие фрагменты на языках разработки сценариев. На самом низком уровне организации кода языки C и C++ почти ничем не отличаются, так что программы на C являются в то же время и программами на C++. Языки C++ и Java являются прямыми потомками C, поэтому в значительной мере совпадают с ним по синтаксису и предлагают столь же эффективные и выразительные средства, в то же время обладая более развитыми наборами типов и библиотек. В нашей повседневной работе мы широко используем все три языка, а также множество других. Выбор языка зависит от конкретной задачи: операционные системы лучше всего писать на скоростном высокоэффективном языке, лишенном всяких ограничений, таком как C или C++; короткие одноразовые задачки прекрасно решаются с помощью языков разработки сценариев или командных интерпретаторов наподобие Awk или Perl; для реализации пользовательских интерфейсов практически нет равных языкам Visual Basic, Tcl/Tk и Java.

Выбор языка для примеров представляет собой нетривиальную методическую задачу. Точно так же, как ни один язык не может справиться со всеми задачами одинаково хорошо, нельзя и проиллюстрировать все темы программирования на одном и том же языке с одинаковой выразительностью. Языки высокого уровня непосредственно диктуют программисту ряд технических решений. На более низком уровне необходимо выбирать между альтернативными подходами к решению. Учитывая больше деталей, можно обсудить вопрос более подробно. Опыт показывает, что даже если мы пользуемся только средствами языков высокого уровня, всегда бывает

полезно знать, как они соотносятся со средствами более низкого уровня. Не обладая этими знаниями, легко наткнуться на непреодолимый барьер в виде непонятного поведения программы или необъяснимого падения быстродействия. Поэтому для наших примеров мы часто выбирали C, хотя на практике предпочли бы какой-нибудь другой язык.

Тем не менее в основном наше изложение можно считать независимым от конкретного языка программирования. Выбор структуры данных диктуется тем языком, который есть “под рукой”; одни языки не предоставляют почти никакого выбора, тогда как другие предлагают широкий круг альтернатив. Но критерии, в соответствии с которыми делается выбор, должны оставаться одними и теми же. Так, мелкие подробности процедур тестирования или отладки в разных языках сильно отличаются, а вот стратегия и тактические приемы практически одинаковы. Бóльшая часть способов и рекомендаций, как сделать программу эффективнее, одинаково применима во всех языках.

На каком бы языке ни писал программист, его задача состоит в том, чтобы выжать максимальный эффект из имеющихся у него в наличии средств. Хороший программист способен перебороть недостатки как неуклюжего языка, так и неповоротливой операционной системы. В то же время даже самая лучшая среда программирования не спасет плохого программиста. Мы надеемся, что эта книга поможет вам программировать лучше и получать больше удовольствия от этой работы независимо от нынешнего уровня вашей квалификации.

Мы хотели бы выразить искреннюю благодарность нашим друзьям и коллегам, которые прочитали рукопись книги и высказали множество ценных советов и пожеланий. Джон Бентли (Jon Bentley), Расс Кокс (Russ Cox), Джон Лакош (John Lakos), Джон Линдерман (John Linderman), Питер Мемишьян (Peter Memishian), Ян Лэнс Тейлор (Ian Lance Taylor), Ховард Трики (Howard Trickey) и Крис Ван Вик (Chris Van Wyk) прочли рукопись, причем неоднократно, с необыкновенной тщательностью и дотошностью. Мы испытываем глубокую признательность к таким людям, как Том Карджилл (Tom Cargill), Крис Клиленд (Chris Cleeland), Стив Дьюхерст (Steve Dewhurst), Эрик Гросс (Eric Grosse), Эндрю Херрон (Andrew Herron), Джерард Хольцман (Gerard Holzmann), Даг Макилрой (Doug McIlroy), Пол Макнами (Paul McNamee), Питер Нельсон (Peter Nelson), Деннис Ритчи (Dennis Ritchie), Рич Стивенс (Rich Stevens), Том Шимански (Tom Szymanski), Кентаро Тояма (Kentaro Toyama), Джон Уэйт (John Wait), Дэниел Ван (Daniel C. Wang), Питер Вайнбергер (Peter Weinberger), Маргарет Райт (Margaret Wright) и Клифф Янг (Cliff Young), за их бесценные комментарии на различных стадиях производства книги. Мы также благодарим за добрый совет и вдумчивые предложения таких лиц, как Эл Ахо (Al Aho), Кен Арнольд (Ken Arnold), Чак Бигелоу (Chuck Bigelow), Джошуа Блох (Joshua Bloch), Билл Кафрэн (Bill Coughran), Боб Фландрена (Bob Flandrena), Рене Франш (Renée French), Марк Керниган (Mark Kernighan), Энди Кёниг (Andy Koenig), Сэйп Мюллендер (Sape Mullender), Эви Немет (Evi Nemeth), Марти Рабинович (Marty Rabinowitz), Марк Шэйни (Mark V. Shaney), Бьорн Страуструп (Bjarne Stroustrup), Кен Томпсон (Ken Thompson) и Фил Уодлер (Phil Wadler). Спасибо всем вам!

Брайан Керниган

Роб Пайк

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес. Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com

WWW: http://www.williamspublishing.com

Информация для писем из:

России: 115419, Москва, а/я 783

Украины: 03150, Киев, а/я 152

1

Стиль программирования

Давно замечено, что лучшие писатели часто пренебрегают законами литературной стилистики. Правда, в каждом случае подобного нарушения читатель обычно получает компенсацию в виде других достоинств литературного произведения. Если же писатель не уверен, что он может предложить что-то взамен, ему лучше придерживаться общих правил стиля.

Уильям Странк, Э.Б. Уайт. “Основы стилистики”
(William Strunk, E.B. White, The Elements of Style*)*

Вот фрагмент кода, взятый из одной большой программы многолетней давности:

```
if   (country    SING) || (country == BRNI) ||
     (country == POL) || (country == ITALY)

        /*
         * Если страна - Сингапур, Бруней или Польша,
         * то ответом будет текущее время, а не
         * записанное.
         * Переустанавливаем время и день недели.
         */
```

Этот фрагмент написан, отформатирован и прокомментирован достаточно тщательно. Да и программа, из которой он взят, работает исключительно хорошо. Программисты, которые ее написали, могут по праву гордиться своим творением. Но данный отрывок все же малопонятен. Какая связь между Сингапуром, Брунеем, Польшей и Италией? Почему Италия не упоминается в комментариях? Раз комментарий и код не согласованы, что-то из них должно быть неправильным (возможно, и то и другое). Код обычно тестируется и запускается на выполнение, поэтому правильным скорее является он, а не комментарий. Возможно, комментарий просто вовремя не обновили при исправлении кода. В комментарии недостаточно говорится о связи между тремя упоминаемыми странами; если этот код придется дорабатывать, нужно будет узнать об этой связи побольше.

Приведенные несколько строк достаточно типичны для реального кода: в основном все написано правильно, но еще есть над чем поработать в плане улучшения.

Наша книга посвящена практическому программированию — как писать реальные, практические программы. Наша цель — помочь вам разрабатывать программы, работающие по крайней мере так же хорошо, как та, из которой взят пример, при

этом избегая сомнительных или потенциально опасных конструкций. Поговорим о том, как же написать хороший код с самого начала, а потом еще и улучшить его по мере доработки. Тем не менее начнем мы с не совсем обычной стартовой точки — стиля программирования. Предназначение стиля программирования состоит в том, чтобы сделать программу легкой для чтения ее автором и другими программистами. Хороший стиль играет решающую роль для достижения хорошего качества программ. Мы обсудим вопросы стиля в самом начале, чтобы читатель имел совершенно ясное понятие об этом при чтении всех последующих глав.

Чтобы написать программу, недостаточно просто соблюсти требования синтаксиса, исправить ошибки и добиться нужного быстродействия. Программы предназначены для чтения не только компьютерами, но и программистами. Хорошо написанную программу легче читать и дорабатывать, чем плохо написанную. Соблюдение стилистической дисциплины дает большую вероятность, что написанный код окажется семантически правильным. К счастью, научиться этой дисциплине совсем несложно.

Принципы стиля программирования основаны на здравом смысле и практическом опыте, а не на взятых с потолка правилах и предписаниях. Код должен быть простым и понятным; в нем должны присутствовать такие черты, как очевидность логики, естественные выражения, типовые конструкции языка, понятные имена, аккуратное форматирование, содержательные комментарии. Хитроумных трюков и необычных конструкций следует избегать. Важно соблюдать однородность, постоянство стиля, чтобы другим программистам было нетрудно разобраться в вашем коде, а вам — в их программах. Это будет легче сделать, если соблюдается единство стиля. Правила стилистики часто задаются принудительно — директивными указаниями свыше или принятым в организации стандартом. Но даже если это не так, всегда лучше соблюдать некий общепринятый стандарт. В этой книге мы придерживаемся стиля, описанного в книге *Язык программирования C* (*The C Programming Language*), с некоторыми незначительными поправками для языков C++ и Java.

Законы стилистики часто будут иллюстрироваться на небольших примерах плохого и хорошего стиля, поскольку контраст между тем, как надо и как не надо писать, весьма поучителен. Эти примеры не взяты с потолка. Так, примеры "как не надо писать" адаптированы из реального кода, написанного обычными программистами (иногда и нами самими) в условиях прессинга, когда нужно было сделать слишком много работы за слишком короткое время. Некоторые из примеров сокращены для удобства, но содержание их ни в коем случае не искажается. Затем примеры переписываются в другой форме, чтобы продемонстрировать пути их усовершенствования. Но поскольку эти фрагменты взяты из реального кода, они могут содержать несколько уровней или классов ошибок и недоработок. Обсуждение и исправление всех этих недостатков заняло бы слишком много места и времени, поэтому даже в примерах "как надо писать" могут по-прежнему скрываться проблемы.

Чтобы отличать плохие примеры от хороших, по всей книге мы сопровождаем строки некачественного кода вопросительными знаками на полях, как в этом отрывке:

```
?    #define ONE 1
?    #define TEN 10
?    #define TWENTY 20
```

В чем же сомнительность данного фрагмента? Подумайте о модификациях, которые придется внести в код, если массив из TWENTY элементов необходимо будет удлинить. Уж по крайней мере стоило бы заменить каждое из символических имен на такое, которое обозначало бы роль соответствующей константы в программе:

```
#define INPUT_MODE 1
#define INPUT_BUFSIZE 10
#define OUTPUT_BUFSIZE 20
```

1.1. Имена

Какой смысл заключен в именах, или идентификаторах? Имя переменной или функции обозначает некий объект и должно нести информацию о предназначении этого объекта. Имя должно быть содержательным, кратким, легко запоминающимся, а если возможно, то и легко произносимым. Много информации можно извлечь из контекста и области действия имени; поэтому, чем шире область действия, тем подробнее должно быть имя.

Пользуйтесь длинными, содержательными именами для глобальных объектов и короткими — для локальных. По определению глобальные имена могут использоваться в любом месте программы, поэтому они должны иметь достаточно описательные и длинные имена, чтобы напоминать читателю об их назначении. Объявление каждого глобального имени полезно сопровождать кратким комментарием:

```
int npending = 0;  // текущая длина входной очереди
```

Глобальные функции, классы и структуры также должны иметь описательные имена, отражающие их роль в программе.

В противоположность этому для локальных переменных подходят и короткие идентификаторы. Например, для локальной переменной внутри функции имени `n` вполне достаточно, `npoints` — тоже неплохо, а вот `numberOfPoints` — уже явный перебор.

Локальные переменные, используемые стандартным образом, вполне могут иметь очень короткие имена. Например, традиционно счетчики циклов обозначаются `i` и `j`, указатели — `p` и `q`, а строки — `s` и `t`. Используя вместо этих обозначений более длинные, можно не только ничего не выиграть, но даже проиграть. Рассмотрим такой пример:

```
?    for (theElementIndex = 0; theElementIndex < numberOfElements;
?            theElementIndex++)
?        elementArray[theElementIndex] = theElementIndex;
```

А теперь перепишем его в следующем виде:

```
for (i = 0; i < nelems; i++)
    elem[i] = i;
```

Программистов часто приучают применять длинные имена вне зависимости от контекста. Это ошибка — удобочитаемость часто достигается именно краткостью записи.

Существует много стандартов именования и различных местных обычаев. Среди самых распространенных — использование имен указателей, начинающихся или заканчивающихся на `p` (таких как `nodep`); написание идентификаторов с прописных букв (`Globals`); написание имен прописными буквами (`CONSTANTS`). В некоторых организациях используются более сложные системы, например, способы кодирования в имени переменной ее типа и употребления. В этом случае `pch` может значить указатель на символ (сокращение от *pointer-to-character*), а `strTo` и `strFrom` — строки, используемые соответственно для записи и чтения. Что касается сокращений и сочетаний внутри имен, то выбор между `npending`, `numPending` или `num_pending` является делом вкуса. Не так важна конкретная система правил, как единообразие стиля и его постоянная привязка к некоему разумному стандарту.

Следование соглашениям об именовании помогает лучше понимать как свой собственный код, так и написанный кем-то другим. Наличие стандарта позволяет легко придумывать новые имена по мере дописывания программы. Чем длиннее программа, тем важнее становится выбор качественных, содержательных, систематических имен.

Пространства имен в C++ и пакеты в Java позволяют управлять областью действия имен и поддерживать их удобочитаемость без необходимости придумывать слишком длинные идентификаторы.

Соблюдайте единообразие и согласованность. Аналогичным или родственным объектам нужно давать аналогичные имена, которые бы демонстрировали родство и подчеркивали различие.

В следующем классе Java имена не просто слишком длинные, они еще и совершенно не согласованы между собой:

```
?     class UserQueue {
?         int noOfItemsInQ, frontOfTheQueue, queueCapacity;
?         public int noOfUsersInQueue() {...}
?     }
```

Слово "queue" ("очередь") фигурирует в идентификаторах как `Q`, `Queue` и `queue`. Но, поскольку обращение к очередям все равно возможно только через переменные типа `UserQueue`, в именах членов этого класса вообще не нужно упоминать слово "queue". Вполне достаточно контекста. Например, следующая запись несет избыточную, дублирующуюся информацию:

```
?    queue.queueCapacity
```

Вот такая версия объявления класса намного лучше:

```
class UserQueue {
    int nitems, front, capacity;
    public int nusers() {...}
}
```

В этом случае операторы работы с очередью будут записываться удобнее:

```
queue.capacity++;
n = queue.nusers();
```

Смысл объявляемых переменных остался полностью понятным. Тем не менее и в этом случае еще есть над чем работать. По сути, `items` ("элементы") и `users` ("пользователи") в данном классе обозначают одно и то же, поэтому следовало бы употребить только один из этих синонимов.

Давайте функциям активные имена. Желательно, чтобы имя функции содержало активный глагол — возможно, с последующим существительным. Например:

```
now = date.getTime();
putchar('\n');
```

Функции, возвращающие логические значения (ИСТИНА или ЛОЖЬ), должны носить такие имена, чтобы было однозначно понятно, в каком случае возвращается каждое из возможных значений. Например, в этой записи непонятно, когда возвращается ИСТИНА, а когда — ЛОЖЬ:

```
?   if (checkoctal(c)) ...
```

Дадим этой функции другое имя:

```
    if (isoctal(c)) ...
```

Теперь ясно, что функция возвращает значение ИСТИНА, если ее аргумент является восьмеричным, и ЛОЖЬ в противоположном случае.

Будьте пунктуальны. Имя не только обозначает объект, оно еще и несет информацию читателю. Дезориентирующее имя может помешать программисту найти в программе ошибку.

Один из авторов в свое время написал макрос под названием `isoctal`, который много лет распространялся в составе библиотек языка C:

```
?   #define isoctal(c) ((c)>= '0' && (c) <= '8')
```

В этом макросе есть ошибка. Правильно было бы написать так:

```
    #define isoctal(c) ((c)>= '0' && (c) <= '7')
```

В данном случае имя макроса правильно выражает намерение автора, но его реализация оказалась ошибочной. Разумный выбор имени объекта запросто может скрыть от внимания разработчика недоработки в самом объекте.

Вот пример, в котором имя и код метода находятся в противоречии друг к другу:

```
?   public boolean inTable(Object obj) {
?       int j = this.getIndex(obj);
?       return (j == nTable);
?   }
```

Функция `getIndex` возвращает значение из диапазона от `0` до `nTable-1`, если она находит объект в таблице, и значение `nTable`, если не находит. Таким образом, декларируемое логическое значение, которое возвращает функция `inTable`, на самом деле выражает совсем не то, что подразумевается ее именем. Во время написания программы с участием этой функции, возможно, ничего страшного не произошло, но если впоследствии кто-то другой решит доработать программу или использовать ее, имя функции наверняка вызовет путаницу.

Упражнение 1.1. Прокомментируйте выбор имен и значений в следующем фрагменте кода:

```
?    #define TRUE 0
?    #define FALSE 1
?
?    if ((ch = getchar()) == EOF)
?        not_eof = FALSE
```

Упражнение 1.2. Доработайте эту функцию:

```
?    int smaller(char *s, char *t) {
?        if (strcmp(s, t) < 1)
?            return 1;
?        else
?            return 0;
?    }
```

Упражнение 1.3. Прочитайте этот код вслух:

```
?    if ((falloc(SMRHSHSCRTCH, S_IFEXT|0644, MAXRODDHSH)) < 0)
?        ...
```

1.2. Выражения и операторы

По аналогии с тем, как следует выбирать идентификаторы для лучшего понимания программы, выражения и операторы тоже необходимо составлять так, чтобы по возможности исключить сложность или двусмысленность восприятия. Из всех вариантов, которые решают поставленную задачу, выбирайте самый понятный. Для группировки операндов окружайте знаки операций пробелами; или, обобщая эту рекомендацию, используйте визуальное форматирование для улучшения удобочитаемости. Это просто, но очень полезно, наподобие того, как чисто убранный рабочий стол позволяет легко находить на нем нужные вещи. Но рабочий стол — это ваше личное дело, а вот написанные вами программы скорее всего будут читать другие люди.

Структурируйте операторы с помощью отступов. Последовательное и логичное применение отступов — это способ сделать структуру программы проще и очевиднее ценой самых минимальных усилий. Ниже приведен пример плохого форматирования кода.

```
?    for(n++;n<100;field[n++]='\0');
?    *i = '\0'; return('\n');
```

Попробуем несколько улучшить читаемость этого фрагмента, переформатировав его:

```
?    for (n++; n < 100; field[n++] = '\0')
?        ;
?    *i = '\0';
?    return('\n');
```

Еще лучше будет поместить присваивание в тело цикла и отделить инкрементирование счетчика, чтобы цикл принял более стандартную форму и стал понятнее при чтении:

```
for (n++; n < 100; n++)
    field[n] = '\0';
*i = '\0';
return('\n');
```

Записывайте выражения в естественной форме. Следует писать выражения так же, как бы вы читали их вслух. Условные выражения, содержащие операцию отрицания, всегда труднее понять:

```
?     if (!(block_id < actblks) || !(block_id >= unblocks))
?         ...
```

Каждая из проверок в условии записана в форме отрицания, хотя в обоих случаях для этого нет никакой необходимости. Сделаем все наоборот и построим утвердительную форму обеих проверок:

```
if ((block_id >= actblks) || (block_id < unblocks))
    ...
```

Теперь код выглядит более естественно.

Используйте скобки во избежание двусмысленности. Скобки существуют для того, чтобы группировать операнды. С их помощью можно более явно обозначить выполняемую операцию даже тогда, когда синтаксис этого не требует. В предыдущем примере внутренние скобки необязательны, однако и вреда от них никакого. Бывалые программисты, наверное, опустили бы эти скобки, потому что операции сравнения (`< <= == != >= >`) имеют более высокий приоритет, чем логические операции (`&&` и `||`). И все же при совместном использовании операций с различным приоритетом лучше не жалеть скобок. Язык С и его “родственники” чреваты опасными ошибками, возникающими из-за приоритета операций. Такие ошибки сделать легко, а обнаружить трудно. Поскольку логические операции представляют более тесную связь между операндами, чем оператор присваивания, скобки обязательны практически во всех выражениях, где эти знаки используются совместно:

```
while ((c = getchar()) != EOF)
    ...
```

Поразрядные двоичные операции `&` и `|` имеют более низкий приоритет, чем операции сравнения наподобие `==`, так что не обманывайтесь внешним видом следующего выражения:

```
?     if (x&MASK == BITS)
?         ...
```

Это выражение означает не то, что можно подумать из-за пространственной группировки операндов, а нечто другое:

```
?     if (x & (MASK == BITS))
?         ...
```

А это явно не входило в намерения программиста. Поскольку в выражении сочетаются поразрядные операции и операции сравнения, необходимо использовать скобки:

```
if ((x&MASK) == BITS)
    ...
```

Даже если в скобках нет необходимости, они могут помочь сориентироваться в таком выражении, группировка операндов в котором не ясна с первого взгляда. Например, в следующем фрагменте синтаксис языка не требует применения скобок:

```
?    leap_year = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

Тем не менее скобки делают выражение более удобочитаемым:

```
leap_year = ((y % 4 == 0) && (y % 100 != 0)) || (y % 400 == 0);
```

В последнем выражении по сравнению с первоначальным удалось также убрать некоторое количество пробелов. Группировка операндов, относящихся к операциям с высоким приоритетом, помогает лучше рассмотреть структуру всего выражения.

Разбивайте сложные выражения на части. Языки C, C++ и Java обладают развитым синтаксисом выражений, поэтому программисты часто соблазняются возможностью "впихнуть" в одно выражение побольше операций. Например, следующее выражение довольно компактно, однако для одного оператора в нем слишком много составных элементов:

```
?    *x += (*xp=(2*k < (n-m) ? c[k+1] : d[k--]));
```

Это выражение воспринимается лучше, если разбить его на несколько частей:

```
if (2*k < n-m)
    *xp = c[k+1];
else
    *xp = d[k--];
*x += *xp;
```

Выражайтесь как можно понятнее. Программисты прилагают нескончаемые усилия, чтобы написать как можно более краткий код или изобрести самый остроумный способ достичь нужного результата. Правда, часто эти усилия направлены не на ту цель — ведь необходимо писать самый понятный, а не самый изощренный код.

Вот, например, какую операцию выполняет этот хитроумный оператор?

```
?    subkey = subkey >> (bitoff - ((bitoff >> 3) << 3));
```

Во внутреннем выражении значение `bitoff` сдвигается на три бита вправо. Затем результат сдвигается влево, и тем самым три сдвинутых бита заменяются нулями. Этот результат вычитается из исходного значения, и все это в итоге дает три младших бита переменной `bitoff`. Полученные три бита используются для сдвига переменной `subkey` вправо.

Таким образом, первоначальное выражение эквивалентно следующему:

```
subkey = subkey >> (bitoff & 0x7);
```

Некоторые конструкции, кажется, просто обречены на злоупотребление. Именно такова операция выбора по условию `?:`, с помощью которой можно построить довольно загадочные выражения:

```
?   child=(!LC&&!RC)?0:(!LC?RC:LC);
```

Почти невозможно догадаться, что же все-таки делает этот оператор, пока не проследишь все возможные альтернативы выбора. Следующая форма значительно длиннее, но понять ее гораздо легче, потому что эти альтернативы более очевидны:

```
if (LC == 0 && RC == 0)
    child = 0;
else if (LC == 0)
    child = RC;
else
    child = LC;
```

Операция `?:` хороша для коротких выражений, в которых она может заменить три-четыре строки операторов `if-else`. Вот пример:

```
max = (a > b) ? a : b;
```

И еще один пример:

```
printf("The list has %d item%s\n", n, n==1 ? "" : "s");
```

Но в целом операцию выбора по условию нельзя считать равноценным заменителем условного оператора.

Понятность кода — это совсем не то же самое, что его краткость. Довольно часто более понятный код является в то же время и более кратким — как, например, в приведенном выше операторе со сдвигом. С другой стороны, он может быть и более длинным, как в примере с заменой операции выбора по условию операторами `if-else`. Единственным критерием выбора тут является легкость понимания кода.

Учитывайте возможные побочные эффекты. Такие операции, как инкрементирование `++`, имеют *побочные эффекты*: кроме возвращения определенного значения, они еще и изменяют значение своего аргумента. Эти побочные эффекты бывают очень удобны, однако из-за них могут также возникнуть проблемы, поскольку операции извлечения значения и модификации переменной иногда не совпадают по времени. В языках C и C++ порядок выполнения операций, составляющих побочные эффекты, не определен, поэтому при множественном присваивании результат может оказаться неправильным:

```
?   str[i++] = str[i++] = ' ';
```

В намерения программиста входило помещение пробелов в следующие две позиции массива `str`. Но в зависимости от того, когда модифицируется индекс `i`, одна из позиций `str` может оказаться пропущенной, и индекс в итоге увеличится только на единицу, а не на два. Следует разбить этот оператор на две части:

```
str[i++] = ' ';
str[i++] = ' ';
```

Даже если оператор содержит всего одно инкрементирование, он все равно может дать непредсказуемый результат. Рассмотрим следующий пример:

```
?    array[i++] = i;
```

Если, например, вначале переменная `i` была равна 3, то элемент массива может получить значение как 3, так и 4.

Побочными эффектами обладают не только операции инкрементирования и декрементирования. Еще один источник скрытых проблем — это операции ввода-вывода. Вот пример попытки чтения двух взаимосвязанных значений из стандартного потока ввода:

```
?    scanf("%d %d", &yr, &profit[yr]);
```

Это некорректный оператор, потому что в одной его части переменная `yr` модифицируется, а в другой — сразу же используется. Значение `profit[yr]` практически не имеет шансов оказаться правильным, разве только в том случае, если старое и новое значения `yr` совпадают. Можно было бы подумать, что результат зависит от порядка вычисления аргументов, но проблема-то как раз в том, что *все* аргументы функции `scanf` вычисляются до ее вызова, т.е. адрес `&profit[yr]` всегда вычисляется с использованием старого значения `yr`. Подобная проблема может возникнуть практически в любом языке программирования. Ее решение, как и в одном из предыдущих примеров, состоит в том, чтобы разбить один оператор на два:

```
     scanf("%d", &yr);
     scanf("%d", &profit[yr]);
```

Следует соблюдать максимальную осторожность при работе с любым выражением, содержащим побочные эффекты.

Упражнение 1.4. Усовершенствуйте все приведенные ниже фрагменты кода.

```
?    if ( !(c == 'y' || c == 'y') )
?        return;

?    length = (length < BUFSIZE) ? length : BUFSIZE;

?    flag = flag ? 0 : 1;

?    quote = (*line == '"') ? 1 : 0;

?    if (val & 1)
?        bit = 1;
?    else
?        bit = 0;
```

Упражнение 1.5. Найдите ошибки или недочеты в следующем фрагменте кода:

```
?    int read(int *ip) {
?        scanf("%d", ip);
?        return *ip;
?    }
?        ...
?    insert(&graph[vert], read(&val), read(&ch));
```

Упражнение 1.6. Перечислите все различные результаты, которые может сгенерировать данный фрагмент кода при разном порядке вычисления выражений:

```
?    n = 1;
?    printf("%d %d\n", n++, n++);
```

Протестируйте этот фрагмент на как можно большем количестве разных компиляторов, чтобы посмотреть, что же может произойти на практике.

1.3. Единообразие стиля и устойчивые конструкции

Соблюдение единообразного стиля позволяет писать программы более высокого качества. Если форматирование кода непредсказуемо меняется от одного блока к другому, или перебор массива в цикле выполняется то в одном, то в другом порядке, или строки копируются один раз функцией `strcpy`, а другой раз — в цикле `for`, то все эти вариации очень сильно мешают правильному восприятию программы и пониманию того, что же она действительно делает. А вот если одна и та же вычислительная операция выполняется каждый раз одинаково, то какие-либо отличия появляются только в том случае, если изменяется и сам характер выполняемых вычислений, что позволяет легко привлечь внимание программиста.

Используйте единую систему отступов и расстановки фигурных скобок. Отступы призваны наглядно демонстрировать структуру программы. Какой же стиль абзацного отступа наилучший? Следует ли ставить открывающую фигурную скобку в той же строке, что и оператор `if`, или в следующей? Программисты постоянно спорят о форматировании исходного кода, но сам по себе стиль не так важен, как его строгое и последовательное соблюдение. Выберите один стиль (лучше всего — наш), придерживайтесь его во всех своих программах, и хватит тратить время на ненужные споры.

Следует ли ставить фигурные скобки в том случае, если синтаксис этого не требует? Дополнительные фигурные скобки, как и квадратные, могут помочь устранить двусмысленность и сделать код понятнее. Для единообразия стиля многие программисты всегда заключают тела циклов или операторов `if` в скобки. Если тело состоит из одного оператора, то в скобках нет необходимости и их часто опускают. Если вы также опускаете лишние скобки, то по крайней мере сохраняйте их в том случае, когда они необходимы для разрешения неоднозначности “висячего” блока `else`. Эту неоднозначность можно проиллюстрировать на следующем примере:

```
?    if (month == FEB) {
?         if (year%4 == 0)
?              if (day > 29)
?                   legal = FALSE;
?         else
?              if (day > 28)
?                   legal = FALSE;
?    }
```

В данном случае отступ вводит в заблуждение, поскольку блок `else` на самом деле относится к строке

```
?               if (day > 29)
```

Таким образом, в коде сделана ошибка. В тех случаях, когда один оператор `if` следует сразу за другим, всегда используйте дополнительные скобки:

```
?    if (month == FEB) {
?         if (year%4 == 0) {
?              if (day > 29)
?                   legal = FALSE;
?         } else {
?              if (day > 28)
?                   legal = FALSE;
?         }
?    }
```

При использовании специальных редакторов со встроенными средствами анализа и коррекции синтаксиса такие ошибки становятся менее вероятными.

Хотя ошибка исправлена, читать этот код все еще трудно. Проследить за выполняемыми вычислениями будет легче, если ввести специальную переменную для хранения количества дней в феврале:

```
?    if (month == FEB) {
?        int nday;
?
?        nday = 28;
?        if (year%4 == 0)
?            nday = 29;
?        if (day > nday)
?            legal = FALSE;
?    }
```

Получившийся код все еще работает неправильно. Например, 2000 год является високосным, тогда как 1900 и 2100 — нет, хотя программа сочтет их високосными. Но этот вариант уже гораздо легче довести до полного совершенства, чем предыдущие.

Кстати, если вы дорабатываете программу, написанную не вами, лучше придерживайтесь того стиля, который вы в ней обнаружите. Внося изменения, не прибегайте к своему собственному стилю, даже если он вам больше нравится. Единообразие стиля все-таки важнее, потому что следующим доработчикам после вас так будет легче работать.

Используйте устойчивые конструкции для поддержания единого стиля. Как и в человеческих языках, в языках программирования существуют устойчивые конструкции (*идиомы*) — стандартные варианты реализации самых распространенных операций. Главное в изучении любого языка — это как следует ознакомиться с системой его идиом.

К числу наиболее распространенных идиом принадлежат формы записи циклов. Рассмотрим код на языках C, C++ и Java, реализующий перебор массива из `n` элементов. Такой перебор может понадобиться, например, для инициализации этого массива. Можно написать нечто подобное:

```
?   i = 0;
?   while (i <= n-1)
?       array[i++] = 1.0;
```

Возможен и такой вариант:

```
?   for (i = 0; i < n; )
?       array[i++] = 1.0;
```

Не исключена также подобная форма записи:

```
?   for (i = n; --i >= 0; )
?       array[i] = 1.0;
```

Все эти формы правильны, но идиоматической, устойчивой конструкцией является только следующая:

```
    for (i = 0; i < n; i++)
        array[i] = 1.0;
```

Именно эта запись выбрана в качестве идиомы не случайно. Во-первых, в ней по очереди перебираются все `n` элементов массива по порядку от `0` до `n-1`. Во-вторых, все управление циклом сосредоточено в его заголовке. В-третьих, операция выполняется в естественном порядке — по возрастанию, — причем для приращения счетчика цикла используется такое стандартное, общеизвестное средство, как операция инкрементирования. В-четвертых, после выполнения цикла индексная переменная остается равной удобному известному значению — количеству элементов в массиве, или индексу следующего за последним элемента. Опытные знатоки языка распознают эту конструкцию так же мгновенно и без всяких раздумий, как и напишут ее.

В языках C++ и Java широко распространен вариант с объявлением счетчика цикла прямо в заголовке:

```
    for (int i = 0; i < n; i++)
        array[i] = 1.0;
```

А вот стандартный цикл для перебора списка в языке C:

```
    for (p = list; p != NULL; p = p->next)
        ...
```

И в этом случае все управление циклом сосредоточено в заголовке `for`.

Для организации бесконечного цикла мы предпочитаем использовать такую конструкцию:

```
    for (;;)
        ...
```

Впрочем, популярностью пользуется и эта форма:

```
    while (1)
        ...
```

Никаких других конструкций, кроме этих двух, использовать не следует.

Система абзацных отступов тоже должна быть стандартной, по существу идиоматической. Например, следующая вертикальная (совершенно необычная) запись цикла портит все впечатление — вся конструкция кажется вовсе не циклом, а тремя отдельными операторами:

```
?    for (
?        ap = arr;
?        ap < arr + 128;
?        *ap++ = 0
?        )
?    {
?      ;
?    }
```

Стандартный цикл читать гораздо легче:

```
     for (ap = arr; ap < arr+128; ap++)
         *ap = 0;
```

К тому же слишком сильно разнесенный по строкам код не отличается компактностью и не помещается на экран или страницу. Это неудобно для его чтения.

Часто встречается такая устойчивая конструкция, как помещение присваивания в условие цикла. Например:

```
     while ((c = getchar()) != EOF)
         putchar(c);
```

Оператор `do-while` используется гораздо реже, чем `for` или `while`, потому что он всегда выполняется как минимум один раз еще до проверки своего условия. Другими словами, условие проверяется в конце цикла, а не в его начале. Во многих случаях эта конструкция представляет собой скрытую ловушку, как, например, в этом варианте цикла считывания и записи символов:

```
?    do {
?        c = getchar();
?        putchar(c);
?    } while (c != EOF);
```

В поток вывода легко может попасть лишний символ, потому что проверка конца файла выполняется после вызова функции `putchar`. Циклом `do-while` следует пользоваться только тогда, когда его тело обязательно должно выполняться хотя бы один раз. Позже будут рассмотрены некоторые примеры.

Одно из преимуществ единообразия в использовании идиом состоит в том, что нестандартные (а значит, потенциально ошибочные) циклы сразу привлекают к себе внимание:

```
?    int i, *iArray, nmemb;
?
?    iArray = malloc(nmemb * sizeof(int));
?    for (i = 0; i <= nmemb; i++)
?        iArray[i] = i;
```

При распределении памяти для массива было выделено `nmemb` элементарных ячеек — от `iArray[0]` до `iArray[nmemb-1]`. Но цикл выполняется от `0` до `nmemb`, так что затирается участок памяти непосредственно после конца массива. К сожале-

нию, подобные ошибки часто обнаруживаются только тогда, когда данным уже нанесен существенный вред.

В языках C и C++ имеются устойчивые конструкции для распределения памяти под строки и работы с ними. Код, в котором не используются такие конструкции, часто бывает написан с ошибками:

```
?   char *p, buf[256];
?
?   gets(buf);
?   p = malloc(strlen(buf));
?   strcpy(p, buf);
```

Функции `gets` необходимо избегать, потому что нет способа ограничить считываемый ею объем данных. Отсюда возникают проблемы безопасности, к которым мы вернемся в главе 6. Там мы покажем, что функция `fgets` всегда является лучшим выбором для подобных операций. Но существует и еще одна проблема: функция `strlen` не включает завершающий нуль, `'\0'`, в подсчитанное ею количество символов, тогда как `strcpy` копирует вместе с другими и его тоже. Таким образом, для строки выделяется недостаточно памяти, и функция `strcpy` выполняет запись за пределами отведенного пространства. Стандартная конструкция выглядит таким образом:

```
p = malloc(strlen(buf)+1);
strcpy(p, buf);
```

И еще один вариант, на этот раз в C++:

```
p = new char[strlen(buf)+1];
strcpy(p, buf);
```

Если вы пропустили дополнительную единицу (`+1`), берегитесь!

В языке Java данная конкретная проблема не существует, потому что там строки не представляются массивами символов с завершающими нулями. К тому же индексы массивов подвергаются проверке, поэтому в Java нет возможности выйти за пределы заданных границ массива.

В большинстве сред программирования на C и C++ имеется библиотечная функция `strdup`, создающая копию строки с использованием функций `malloc` и `strcpy`. С ее помощью легко избежать описанной ошибки. К сожалению, функция `strdup` не определена в стандарте ANSI C.

Кстати, ни первоначальная, ни исправленная версия приведенного примера не содержит проверки значения, возвращаемого функцией `malloc`. Мы опустили этот момент, чтобы сосредоточиться на главном, однако в любой реальной программе необходимо проверять значения, возвращаемые из таких функций распределения памяти, как `malloc`, `realloc`, `strdup` и др.

Используйте каскад `else if` для реализации многовариантного выбора. Принятие решений в зависимости от нескольких вариантов выбора описывается такой идиоматической конструкцией, как цепочка `if ... else if ... else`. Вот ее общая структура:

```
if (условие_1)
    оператор_1
else if (условие_2)
    оператор_2
...
else if (условие_n)
    оператор_n
else
    оператор_по_умолчанию
```

Условия проверяются сверху вниз. Как только найдено первое выполняющееся условие, управление передается следующим за ним операторам, а затем остальная часть каскада пропускается. *Операторы* могут быть одиночными или объединенными в блоки — группы в фигурных скобках.

Последний оператор обрабатывает ситуацию "по умолчанию", т.е. случай, когда не выполняется ни одно из условий. Завершающий блок `else` можно опустить, если по умолчанию ничего не нужно делать. Правда, его можно и оставить, включив туда вывод сообщения об ошибке. Это превентивная мера на случай возникновения "невозможной" ситуации, т.е. непредвиденной в программе.

В такой конструкции следует выровнять все ключевые слова `else` по одной вертикальной линии, а не ставить их в соответствие операторам `if`. Вертикальное выравнивание подчеркивает, что выполняется последовательная проверка, и не дает строкам сползать к правому краю страницы.

Длинный каскад вложенных операторов `if` часто свидетельствует о низком качестве кода, а то и об откровенных ошибках в нем:

```
?   if (argc == 3)
?       if ((fin = fopen(argv[1], "r")) != NULL)
?           if ((fout = fopen(argv[2], "w")) != NULL) {
?               while ((c = getc(fin)) != EOF)
?                   putc(c, fout);
?               fclose(fin); fclose(fout);
?           } else
?               printf("Can't open output file %s\n", argv[2]);
?       else
?           printf("Can't open input file %s\n", argv[1]);
?   else
?       printf("Usage: cp inputfile outputfile\n");
```

Этот каскад операторов `if` требует держать в уме "стек" условий и проверок, чтобы в нужных местах доставать из этого стека информацию о выполняемых операциях (если мы еще будем в состоянии их вспомнить). Поскольку в каждом случае выполняется максимум одна операция, следует воспользоваться системой операторов `else if`. Если изменить порядок проверок и принятия решений, код станет более понятным. Кроме того, удастся устранить некоторую утечку ресурсов, имевшую место в исходном варианте:

```
if (argc != 3)
    printf("Usage: cp inputfile outputfile\n");
else if ((fin = fopen(argv[1], "r")) == NULL)
    printf("Can't open input file %s\n", argv[1]);
else if ((fout = fopen(argv[2], "w")) == NULL) {
```

```
        printf("Can't open output file %s\n", argv[2]);
        fclose(fin);
    } else {
        while ((c = getc(fin)) != EOF)
            putc(c, fout);
        fclose(fin);
        fclose(fout);
    }
```

Проверки выполняются одна за другой, пока одна из них не даст положительный результат. Затем выполняются соответствующие операторы, и управление переходит за пределы последнего `else`. Общее правило состоит в том, чтобы немедленно сопровождать каждое принятое решение соответствующими операциями, если это возможно. Говоря более простыми словами, как только сделали выбор — выполните какую-нибудь операцию.

При попытке перенести в новую программу готовый фрагмент кода можно столкнуться с весьма компактной и малопонятной записью, например:

```
?   switch (c) {
?   case '-':  sign = -1;
?   case '+':  c = getchar();
?   case '.':  break;
?   default:   if (!isdigit(c))
?                  return 0;
?   }
```

В этом операторе `switch` используется хитроумно выстроенная последовательность операторов, выполняемая “насквозь” для экономии строк. Она не относится к стандартным, идиоматическим конструкциям, потому что блоки `case` должны практически всегда заканчиваться операторами `break`, а редкие исключения из этого правила должны быть прокомментированы. Более традиционное оформление оператора `switch` длиннее, но зато понятнее.

```
?   switch (c) {
?   case '-':
?       sign = -1;
?       /* выполняется насквозь */
?   case '+':
?       c = getchar();
?       break;
?   case '.':
?       break;
?   default:
?       if (!isdigit(c))
?           return 0;
?   }
```

Удлинение кода более чем компенсируется удобством его чтения. И все-таки для этой нестандартной конструкции еще лучше подошел бы каскад операторов `if-else`:

```
    if (c == '-') {
        sign = -1;
        c = getchar();
```

```
} else if (c == '+') {
    c = getchar();
} else if (c != '.' && !isdigit(c)) {
    return 0;
}
```

Фигурные скобки вокруг блоков, состоящих из одной строки, подчеркивают параллельную структуру алгоритма.

Сквозной проход через блоки `case` вполне приемлем, если в нескольких блоках должны выполняться одни и те же операции. Обычное оформление выглядит так:

```
case '0':
case '1':
case '2':
    ...
    break;
```

В этом случае дополнительный комментарий не требуется.

Упражнение 1.7. Перепишите эти фрагменты кода на языке C/C++ в более удобной форме:

```
?    if(istty(stdin))  ;
?    else if (istty(stdout))  ;
?         else if (istty(stderr))  ;
?              else return(0);


?    if (retval != SUCCESS)
?    {
?        return (retval);
?    }
?    /* Все прошло хорошо */
?    return SUCCESS;


?    for (k = 0; k++ < 5; x += dx)
?        scanf("%lf", &dx);
```

Упражнение 1.8. Найдите ошибки в следующем фрагменте кода на языке Java и исправьте их, переписав код с применением стандартных конструкций:

```
?    int count = 0;
?    while (count < total) {
?        count++;
?        if (this.getName(count) == nametable.userName()) {
?            return(true);
?        }
?    }
```

1.4. Макрофункции

Представители старой школы программирования на С очень любят писать макросы вместо функций для быстрых и коротких вычислений, которые выполняются очень часто. Среди таких макросов имеются даже официально санкционированные для целей ввода-вывода (например, `getchar`) или анализа символов (например, `isdigit`). Поводом для их написания служит повышение быстродействия — с помощью макроса можно избежать потерь времени на вызов функции. Но этот аргумент был не очень убедителен даже в первые годы существования С, когда компьютеры работали медленно, а функции вызывались долго; в настоящее время об этом вообще смешно говорить. При современном уровне развития вычислительной техники и компиляторов недостатки макрофункций с лихвой покрывают то небольшое преимущество, которое достигается отсутствием затрат на вызов.

Избегайте макрофункций. В языке С++ встраиваемые (`inline`) функции сделали функциональные макросы ненужными. В Java макросов нет вообще. Даже в своем родном языке С они создают больше проблем, чем решают.

Одна из наиболее серьезных проблем с макрофункциями состоит в том, что если в ее определении несколько раз фигурирует один и тот же параметр, то он может вычисляться тоже несколько раз. Если макрофункция содержит операции с побочными эффектами, в результате получается трудноуловимая ошибка. Ниже приведен пример, в котором реализуется одна из функций анализа символов из заголовочного файла `<ctype.h>`:

```
?    #define isupper(c) ((c) >= 'A' && (c) <= 'Z')
```

Обратите внимание, что параметр `c` фигурирует в определении макроса дважды. А теперь представим, что эта макрофункция вызывается в таком контексте:

```
?    while (isupper(c = getchar()))
?        ...
```

Теперь всякий раз, когда введенный символ по своему значению больше или равен `'A'`, он будет затираться следующим символом, сравниваемым уже с `'Z'`. Стандарт С проработан достаточно тщательно, что позволяет определить `isupper` и другие аналогичные функции как макросы, но только в том случае, если они гарантируют однократное вычисление аргумента. Поэтому приведенная выше реализация не годится.

Всегда лучше пользоваться готовыми функциями из файла `ctype`, чем писать их самому. В целях безопасности следует также избегать вложенных функций типа `getchar`, имеющих побочные эффекты. Перепишем анализ символа с использованием двух выражений вместо одного. Кроме всего прочего, это позволит еще и не пропустить возможную ситуацию конца файла:

```
     while ((c = getchar()) != EOF && isupper(c))
         ...
```

Иногда многократное вычисление выражения или аргумента приводит к простому ухудшению быстродействия, а не к явной ошибке. Например:

```
?     #define ROUND_TO_INT(x)  ((int) ((x)+(((x)>0)?0.5:-0.5)))
?         ...
?     size = ROUND_TO_INT(sqrt(dx*dx + dy*dy));
```

В этом фрагменте кода квадратный корень вычисляется вдвое чаще, чем это необходимо. Такое сложное выражение, как тело макроса ROUND_TO_INT, пусть даже с простым аргументом, транслируется в длинную последовательность инструкций, которую следовало бы поместить в отдельную функцию и вызывать по мере необходимости. Подстановка макрофункции во все места, в которых она вызывается, делает итоговую скомпилированную программу длиннее. (Встраиваемые функции C++ тоже имеют этот недостаток.)

Заключайте в скобки тело макрофункции и ее аргументы. Если все же вам необходимо определить макрофункцию, делайте это с осторожностью. Макрофункция — это простая текстовая подстановка. Параметры в ее объявлении заменяются фактическими аргументами, и полученный фрагмент вставляется в виде текста на место вызова макроса. Это очень существенное отличие от функций, которое может вызвать ряд проблем. Возьмем, например, следующее выражение:

```
      1 / square(x)
```

Здесь все в порядке, если square(x) — обычная функция. Но пусть это будет макрос, определенный следующим образом:

```
?     #define square(x)    (x) * (x)
```

В этом случае макроподстановка даст следующее ошибочное выражение:

```
?     1 / (x) * (x)
```

Чтобы исправить ошибку, макрос следует переписать таким образом:

```
      #define square(x)    ((x) * (x))
```

Все эти скобки совершенно необходимы. Но даже корректное заключение в скобки все равно не решает проблему многократного вычисления аргументов. Если операция достаточно громоздка или настолько часто встречается, что ее можно сделать стандартным средством, лучше оформите ее как функцию.

В языке C++ аппарат встраиваемых функций позволяет избежать всех этих ошибок и при этом сохраняет все преимущества быстродействия, присущие макрофункциям. Он вполне подходит для определения небольших функций, вычисляющих или устанавливающих какое-нибудь одно значение.

Упражнение 1.9. Какие ошибки или проблемы могут возникнуть при использовании этой макрофункции?

```
?     #define ISDIGIT(c) ((c >= '0') && (c <= '9')) ? 1 : 0
```

1.5. Явные числовые константы

В программах часто встречаются записанные в явном виде числовые константы, для обозначения которых существует термин "магические числа" (*magic numbers*). Это могут быть просто числа в выражениях, размеры массивов, позиции символов, коэффициенты преобразований и т.п.

Присваивайте константам символические имена. Любое число, отличное от 0 или 1, вполне может подпадать под приведенное выше определение, поэтому желательно, чтобы оно имело свое имя. Само по себе число в исходном коде не дает никаких указаний о его происхождении, смысле или степени важности, из-за чего программу становится труднее читать и дорабатывать. Например, следующий отрывок из программы построения гистограмм на текстовом терминале размером 24 строки на 80 столбцов слишком перегружен явными константами и поэтому труден для понимания:

```
?   fac = lim / 20;     /* выбор масштабного коэффициента */
?   if (fac < 1)
?       fac = 1;
?                       /* генерирование гистограммы */
?   for (i = 0, col = 0; i < 27; i++, j++) {
?       col += 3;
?       k = 21 - (let[i] / fac);
?       star = (let[i] == 0) ? ' ' : '*';
?       for (j = k; j < 22; j++)
?           draw(j, col, star);
?   }
?   draw(23, 2, ' ');  /* метка оси x */
?   for (i = 'A'; i <= 'Z'; i++)
?       printf("%c  ", i);
```

Среди прочих данных этот код содержит такие константы, как `20`, `21`, `22`, `23` и `27`. Они явно имеют отношение друг к другу... или нет? Фактически в основе работы этой программы лежат всего лишь три важнейшие константы: 24 (количество строк на экране), 80 (количество столбцов на экране) и 26 (количество букв в алфавите). Но ни одна из этих констант в явном виде не присутствует в коде, из-за чего становится непонятно, откуда взялись остальные.

Присваивая основным числовым константам символические имена, код можно сделать намного понятнее. Сразу обнаруживается, что константа `3` возникла в результате операции `(80-1)/26`, а массив `let` должен содержать 26 элементов, а не 27 (лишний элемент появляется оттого, что координаты на экране нумеруются начиная с единицы, а не с нуля). Сделаем еще пару упрощений и получим следующий результат:

```
enum {
    MINROW   =  1,               /* верхний край */
    MINCOL   =  1,               /* левый край */
    MAXROW   =  24,              /* нижний край (<=) */
    MAXCOL   =  80,              /* правый край (<=) */
    LABELROW = 1,                /* местоположение меток */
    NLET     =  26,              /* длина алфавита */
    HEIGHT   =  MAXROW - 4,      /* высота столбцов гистограммы
```

```
*/
        WIDTH     = (MAXCOL-1)/NLET    /* ширина столбцов гистограммы
*/
    }
        ...
        fac = (lim + HEIGHT-1) / HEIGHT; /* выбор масштабного
коэффициента */
        if (fac < 1);
            fac = 1;
        for (i = 0; i < NLET; i++) {    /* генерирование гистограммы
*/
            if (let[i] == 0)
                continue;
            for (j = HEIGHT - let[i]/fac; j < HEIGHT; j++)
                draw(j+1 + LABELROW, (i+1)*WIDTH, '*');
        }
        draw(MAXROW-1, MINCOL+1, ' ');  /* метка оси x */
        for (i = 'A'; i <= 'Z'; i++)
            printf("%c  ", i);
```

Вот теперь уже гораздо понятнее, что же делается в основном цикле. Он представляет собой стандартную конструкцию с перебором от 0 до NLET, т.е. по всем имеющимся элементам данных. Вызовы функции draw также стали более вразумительными, потому что слова MAXROW и MINCOL напоминают о смысле и порядке следования аргументов. Что еще более важно, теперь эту программу легче адаптировать к новому размеру экрана или данным другого типа. Числовые константы утратили загадочность, а заодно и вся программа стала более понятной.

Определяйте числа как переменные-константы, а не как макросы. Программисты на языке C традиционно определяют символические константы с помощью директивы #define. Препроцессор C — это мощный, но неповоротливый инструмент, а макроопределения несут с собой большую опасность, потому что они изменяют лексическую структуру исходного кода программы. Поэтому желательно действовать стандартными средствами самого языка. В языках C и C++ целочисленные константы можно определить как элементы перечислимого типа в операторе enum, как это было сделано в предыдущем примере. В языке C++ константы любого типа можно определить с помощью ключевого слова const:

```
const int MAXROW = 24, MAXCOL = 80;
```

В языке Java для этого служит ключевое слово final:

```
static final int MAXROW = 24, MAXCOL = 80;
```

В языке C тоже есть ключевое слово const, но такие константы не могут служить границами массивов, поэтому в C следует отдавать предпочтение перечислимому типу (enum).

Используйте символьные константы, а не их числовые коды. Для анализа свойств символьных переменных следует пользоваться функциями из заголовочного файла <ctype.h> или их эквивалентами. Возьмем, например, следующую проверку:

```
?    if (c >= 65 && c <= 90)
?        ...
```

Ее правильность целиком и полностью зависит от конкретной кодировки, т.е. кодового символьного набора. Лучше записать это следующим образом:

```
?    if (c >= 'A' && c <= 'Z')
?        ...
```

Но и эта конструкция может не дать желаемого эффекта, если буквы в наборе идут не подряд или же алфавит включает и другие символы. В языке C или C++ лучше всего воспользоваться библиотечной функцией:

```
     if (isupper(c))
         ...
```

В Java эта же проверка оформляется таким образом:

```
     if (Character.isUpperCase(c))
         ...
```

Нечто подобное происходит с числом 0, которое может фигурировать в программах в самых разных контекстах. Конечно, компилятор всегда приведет его к нужному типу самостоятельно, однако при чтении программы полезно понимать, какую роль играет нуль в том или ином случае. Например, в C для обозначения нулевого указателя следует пользоваться выражениями `(void *)0` или `NULL`, а для представления нулевого байта в конце строки — выражением `'\0'`. Другими словами, не следует писать так:

```
?    str = 0;
?    name[i] = 0;
?    x = 0;
```

Лучше записать это таким образом:

```
     str = NULL;
     name[i] = '\0';
     x = 0.0;
```

Здесь всякий раз используется соответствующая ситуации константа с явно выраженным типом, а символ `0` обозначает только целое число "нуль". Если писать именно так, то в каждом случае будет ясно предназначение константы и программа приобретет черты самодокументированности. Правда, в C++ нулевой указатель повсеместно обозначается просто `0`, а не `NULL`. В языке Java этот вопрос решен лучше всего — введением ключевого слова `null`, обозначающим пустую (ни на что не указывающую) объектную ссылку.

Вычисляйте размер объекта стандартными средствами языка. Не обозначайте размеры объектов данных явными константами — т.е. например, пишите `sizeof(int)` вместо `2` или `4`. Аналогично, лучше записать `sizeof(array[0])`, чем `sizeof(int)`, потому что если тип массива изменится, это позволит сэкономить одно исправление.

Операция `sizeof` часто избавляет от необходимости изобретать лишние имена для констант, которые обозначают размеры массивов. Рассмотрим, например, такую запись:

```
char buf[1024];

fgets(buf, sizeof(buf), stdin);
```

Размер буфера — явная литеральная константа, но, к счастью, она встречается в программе всего один раз, а именно в объявлении массива. Нет никакого смысла изобретать символическое имя для размера локального массива, но вот написать код так, чтобы в нем не пришлось ничего менять при изменении размера массива, определенно имеет смысл.

В объектах-массивах Java имеется поле `length`, которое содержит количество элементов:

```
char buf[] = new char[1024];

for (int i = 0; i < buf.length; i++)
    ...
```

В языках С и C++ нет эквивалента поля `.length`, но если объявление массива доступно в данном блоке программы, то его размер можно вычислить с помощью следующего макроса:

```
#define NELEMS(array) (sizeof(array) / sizeof(array[0]))

double dbuf[100];

for (i = 0; i < NELEMS(dbuf); i++)
    ...
```

Размер массива задается только в одном месте; если впоследствии он изменяется, то код не приходится исправлять. В данном случае проблема многократного вычисления аргумента не имеет места, поскольку выполняемые операции не сопровождаются побочными эффектами. Более того, вычисления фактически выполняются на этапе компиляции. Вот это как раз подходящий случай для использования макроса, потому что именно макрос может сделать то, чего не может функция: вычислить размер массива по его объявлению.

Упражнение 1.10. Как бы вы переписали данные определения, чтобы минимизировать потенциальную возможность ошибки?

```
?    #define FT2METER    0.3048
?    #define METER2FT    3.28084
?    #define MI2FT       5280.0
?    #define MI2KM       1.609344
?    #define SQMI2SQM    2.589988
```

1.6. Комментарии

Комментарии предназначены для того, чтобы программу было легче читать и понимать. Но от них мало толку, если они попросту дублируют то, что и так ясно из текста программы, противоречат содержанию кода или отвлекают читателя пространными, красиво оформленными описаниями. Лучшие комментарии – это те, которые помогают лучше понять код, давая либо краткое описание самых существенных деталей, либо расширенное представление о выполняемых операциях.

Не повторяйте очевидное. В комментариях не должна повторяться информация, очевидная из самого текста программы, например, тот факт, что операция `i++` инкрементирует счетчик `i`. Вот некоторые из наших любимых образчиков бесполезных комментариев:

```
?    /*
?     * Блок default
?     */
?    default:
?        break;


?    /* возвращается значение SUCCESS */
?    return SUCCESS;


?    zerocount++;    /* Инкрементирование счетчика нулевой позиции */


?    /* Инициализация поля "total" полем "number_received" */
?    node->total = node->number_received;
```

Такие комментарии нужно удалять из исходного текста; они просто загромождают программу.

Комментарии должны прояснять что-то такое, что не до конца понятно из самого кода, или же сводить воедино информацию, разбросанную по разным местам программы. Если выполняется некая тонкая операция, комментарий может оказаться необходимым для ее понимания, но если все и так ясно из операторов, то пересказ того же самого своими словами лишен смысла:

```
?    while ((c = getchar()) != EOF && isspace(c))
?        ;                              /* пропуск пробелов */
?    if (c == EOF)                      /* конец файла */
?        type = endoffile;
?    else if (c == '(')                 /* левая скобка */
?        type = leftparen;
?    else if (c == ')')                 /* правая скобка */
?        type = rightparen;
?    else if (c == ';')                 /* точка с запятой */
?        type = semicolon;
?    else if (is_op(c))                 /* знак операции */
?        type = operator;
?    else if (isdigit(c))               /* число */
?        ...
```

Эти комментарии тоже следует убрать из программы, поскольку в ней уже определены достаточно информативные символические константы.

Комментируйте функции и глобальные данные. Конечно, комментарии могут быть и полезными. Например, следует комментировать функции, глобальные переменные, определения констант, поля структур и классов и любые места, где резюме может прояснить смысл кода.

Глобальные переменные имеют тенденцию "выскакивать" то тут, то там в разных местах программы; комментарий к ним служит напоминанием, откуда они взялись. Вот пример из программы к главе 3 этой книги:

```
struct State {  /* список префиксов и суффиксов */
    char    *pref[NPREF];   /* слова-префиксы */
    Suffix  *suf;           /* список суффиксов */
    State   *next;          /* следующий элемент в хэш-таблице */
};
```

Комментарий перед определением функции служит удачным введением к чтению самого кода функции. Если функция не слишком длинная и технически несложная, то достаточно одной строки:

```
// random: возвращает целое число из диапазона [0..r-1]
int random(int r)
{
    return (int)(Math.floor(Math.random()*r));
}
```

Иногда код бывает очень сложным; в частности, таковым может быть реализуемый алгоритм или используемая структура данных. В этом случае читателю может помочь ссылка на первоисточник. Может оказаться ценной также информация о том, почему были выбраны те или иные технические решения. Следующий комментарий предваряет исключительно эффективную реализацию обратного дискретного косинус-преобразования (ДКП), используемую в декодере JPEG-изображений:

```
/*
 * idct: масштабированная целочисленная реализация обратного
 * двумерного дискретного косинус-преобразования с ячейкой 8x8,
 * по алгоритму Чен-Вана (IEEE ASSP-32, с.803-816, август 1984)
 *
 * 32-разрядная целочисленная арифметика (8-разрядные коэффициенты)
 * 11 множителей, 29 добавок на одно ДКП
 *
 * Коэффициенты расширены до 12 бит для совместимости
 * со стандартом IEEE 1180-1990
 */

static void idct(int b[8*8])
{
    ...
}
```

В этом полезном комментарии дана ссылка на источник, кратко описаны используемые данные, указаны сведения о производительности алгоритма, а также отмечено, как и зачем этот алгоритм модифицирован по сравнению с исходным.

Не комментируйте плохой код — переписывайте его. Конечно, следует комментировать все необычные или запутанные места. Но если объем комментариев начинает намного превышать объем кода, то, возможно, сам код требует серьезной доработки. В следующем примере для объяснения одного оператора понадобился длинный маловразумительный комментарий, да еще и отладочная директива условной компиляции:

```
? /* Если "result" равен 0, есть соответствие, возвращаем true (не нуль).
?       В противном случае "result" не нуль, возвращаем false (нуль). */
?
?    #ifdef DEBUG
?    printf("*** isword returns !result = %d\n", !result);
?    fflush(stdout);
?    #endif
?
?    return(!result);
```

Отрицания трудно воспринимаются, поэтому их следует избегать. Малоинформативное имя переменной `result` тоже создает проблемы в понимании текста. Если дать ей описательное имя `matchfound` ("найдено соответствие"), комментарий становится вообще ненужным и оператор вывода тоже приобретает более аккуратный вид:

```
    #ifdef DEBUG
    printf("*** isword returns matchfound = %d\n", matchfound);
    fflush(stdout);
    #endif

    return matchfound;
```

Не противоречьте коду. Большинство комментариев соответствует коду в момент его написания, но по мере исправления ошибок и доработки программы содержание комментариев может все больше отклоняться от исходного текста, если их не обновлять. Это вполне вероятное объяснение для того расхождения между кодом и комментарием, которое было приведено в начале этой главы.

В чем бы ни состояла причина расхождения, комментарий, который не соответствует коду, порождает недоразумения. Бывает, что программисты проводят многие бесполезные часы за отладкой кода только потому, что приняли ошибочный комментарий за чистую монету. Внося изменения в код, не забывайте делать это и с комментариями.

Комментарии должны не только согласовываться с кодом, но и обеспечивать его поддержку. В следующем примере комментарий составлен корректно (он объясняет назначение двух последующих строк), но противоречит фактическому содержанию кода, потому что в комментарии речь идет о символах конца строки, а в коде — о символах пустого пространства:

```
?    time(&now);
?    strcpy(date, ctime(&now));
?    /* убираем символ конца строки, скопированный из ctime */
?    i = 0;
?    while(date[i] >= ' ') i++;
?    date[i] = 0;
```

Одно из напрашивающихся усовершенствований — это переписать код в более стандартной форме:

```
?     time(&now);
?     strcpy(date, ctime(&now));
?     /* убираем символ конца строки, скопированный из ctime */
?     for (i = 0; date[i] != '\n'; i++)
?         ;
?     date[i] = '\0';
```

Теперь код и комментарий согласуются между собой, но их можно еще усовершенствовать, сделав более прямыми. Задача состоит в том, чтобы удалить символ конца строки, который функция `ctime` помещает в последнюю позицию возвращаемой ею строки. Комментарий должен утверждать именно это, и то же самое должно быть ясно из исходного кода:

```
      time(&now);
      strcpy(date, ctime(&now));
      /* ctime добавляет символ конца строки; убираем его */
      date[strlen(date)-1] = '\0';
```

Последний оператор представляет собой идиоматическую конструкцию языка С для удаления последнего символа из строки. Теперь код стал коротким, стандартным, хорошо понятным, а комментарий обеспечивает необходимую информационную поддержку в виде объяснения, зачем нужна эта операция.

Проясняйте, а не запутывайте. Предполагается, что комментарии должны делать трудные места программы более понятными, а не создавать дополнительные препятствия для понимания. Следующий пример соответствует нашим рекомендациям по комментированию функций и объяснению их необычных возможностей. С другой стороны, рассматривается функция `strcmp`, и ее необычные возможности играют второстепенную роль по сравнению с основной задачей — реализацией стандартного, хорошо известного интерфейса.

```
?     int strcmp(char *s1, char *s2)
?     /* функция сравнения строк; возвращает -1, если строка s1 */
?     /* старше s2 в алфавитном порядке; 0, если строки равны; */
?     /* и 1, если s1 младше s2 */
?     {
?         while(*s1==*s2)  {
?             if(*s1=='\0')  return(0);
?             s1++;
?             s2++;
?         }
?         if(*s1>*s2) return(1);
?         return(-1);
?     }
```

Если для объяснения операции требуется больше чем несколько слов, это часто указывает на необходимость усовершенствовать код. В нашем случае код, может быть, и можно усовершенствовать, но главная проблема заключается не в нем, а в комментарии. Комментарий имеет почти такую же длину, как и реализация функции; кроме того, он еще и не вполне понятен (например, что такое "старше"?). Подведем итог. Код функции довольно труден для понимания, но это реализация

стандартной операции, поэтому комментарий должен носить такой характер: давать резюме выполняемых действий и ссылаться на источник, в котором определена данная операция. Вот все, что для этого нужно:

```
    /* strcmp: возвращает < 0, если s1<s2, > 0, если s1>s2, 0, если
равны */
    /*         ANSI C, раздел 4.11.4.2 */
    int strcmp(const char *s1, const char *s2)
    {
        ...
    }
```

Студентов учат, что необходимо комментировать все подряд. От профессиональных программистов часто требуют, чтобы они комментировали весь свой код. Но сам смысл комментирования можно легко потерять, слепо следуя предписаниям. Комментарии предназначены для того, чтобы читатель легко понял те части программы, смысл которых не ясен из самого кода. Старайтесь по возможности писать код, легкий для понимания. Чем лучше это получается, тем меньше требуется комментариев. Хороший код обычно требует меньшего объема комментариев, чем плохой.

Упражнение 1.11. Выскажите свое мнение об этих комментариях.

```
?   void dict::insert(string& w)
?   // возвращает 1, если w есть в словаре, 0 в противном случае


?   if (n > MAX || n % 2 > 0) // проверка на четность


?   // Запись сообщения
?   // Приращение счетчика строк после каждой записанной строки
?
?   void write_message()
?   {
?       // приращение счетчика строк
?       line_number = line_number + 1;
?       fprintf(fout, "%d  %s\n%d  %s\n%d  %s\n",
?           line_number, HEADER,
?           line_number + 1, BODY,
?           line_number + 2, TRAILER);
?       // приращение счетчика строк
?       line_number = line_number + 2;
?   }
```

1.7. Зачем утруждать себя

В этой главе мы рассмотрели некоторые основные вопросы стиля программирования: информативные имена объектов, понятную запись выражений, стандартные управляющие конструкции, удобочитаемость кода и комментариев к нему, важность соблюдения единой, последовательной системы соглашений и идиоматических конструкций для решения всех перечисленных задач. Трудно спорить с тем, что все это благотворно сказывается на коде.

Но зачем вообще беспокоиться о стиле? Кому какое дело, как выглядит исходный код программы, если она работает? Не слишком ли много времени нужно тратить на придание коду должного внешнего вида? Разве правила стиля в конце концов не являются надуманными?

Ответ состоит в том, что хорошо написанный код легко читать и понимать, он почти наверняка содержит меньше ошибок, а также очень часто еще и короче кода, небрежно собранного в кучу и не приведенного к хорошему стилю. В стремлении сдать работоспособную программу к заданному сроку бывает очень легко пожертвовать требованиями стиля, чтобы позаботиться о нем позже. Но эта жертва может дорого обойтись. Некоторые из примеров этой главы показывают, что может случиться, если не уделить достаточно внимания хорошему стилю программирования. Небрежный код — это плохой код. Он не просто неуклюже написан и трудночитаем, он чаще всего содержит серьезные ошибки.

Ключевой вывод состоит в том, что хороший стиль программирования нужно сделать своей второй натурой. Если постоянно думать о стиле при написании кода, а также время от времени пересматривать и улучшать его, то у вас со временем появится новая хорошая привычка. Как только у вас выработается необходимый автоматизм, требования стиля будут удовлетворяться подсознательно, и даже программы, написанные в предельно сжатые сроки, приобретут более высокое качество.

Дополнительная литература

Как говорилось в начале главы, написание хорошего кода имеет много общего с написанием хорошего литературного текста. Лучшей из небольших книг по стилистике английского языка по праву считается книга W. Strunk, E.B. White, *The Elements of Style* (Allyn & Bacon).

В этой главе излагается подход к стилю, принятый в книге B.Kernighan, P.J. Plauger, *The Elements of Programming Style* (McGraw-Hill, 1978). Превосходные рекомендации по стилю программирования можно найти в книге Steve Maguire, *Writing Solid Code* (Microsoft Press, 1993). Полезные дискуссии по стилю имеются также в книгах Steve McConnell, *Code Complete* (Microsoft Press, 1993) и Peter van den Linden, *Expert C Programming: Deep C Secrets* (Prentice Hall, 1994).

2

Алгоритмы и структуры данных

В конечном счете, только знакомство с методами и средствами, применяемыми на практике, может помочь найти правильное решение для поставленной задачи, и только наличие определенного опыта позволит неизменно достигать профессиональных результатов.

Рэймонд Филдинг. "Техника съемки спецэффектов"
(Raymond Fielding, The Technique of Special Effects Cinematography*)*

Изучение алгоритмов и структур данных составляет фундамент науки информатики и программирования; это огромная область знаний, изобилующая элегантными методами и тонкими математическими доказательствами. Теорию алгоритмов и структур данных не следует воспринимать как игрушку для ученых-теоретиков. Хороший алгоритм или структура делает возможным секундное решение задачи, на которую могли бы уйти годы работы.

В таких специализированных областях, как графика, управление базами данных, синтаксический анализ, численные методы, моделирование, способность решать задачи почти целиком зависит от знания разработанных в этой области алгоритмов и структур данных. Если вы пишете программу из области, новой для вас, то вы просто *обязаны* выяснить, что уже наработано другими людьми в этой области. Иначе можно потратить время зря, делая плохо то, что другие давно сделали хорошо.

Алгоритмы и структуры данных используются в каждой программе, но лишь в очень немногих программах приходится изобретать что-то действительно новое. Даже в таких сложных программах, как компиляторы и Web-браузеры, структуры данных представляют собой большей частью массивы, списки, деревья и хэш-таблицы. Если потребуются какие-то более сложные объекты, то почти наверняка они будут основаны на перечисленных простых конструкциях. Соответственно, для большинства программистов задача заключается в том, чтобы узнать, какие алгоритмы и структуры данных существуют для решения задач определенного типа, и правильно выбрать нужные из их числа.

Имеется всего лишь несколько фундаментальных алгоритмов, встречающихся почти во всех программах (в основном это алгоритмы поиска и сортировки), причем они нередко включены в состав библиотек. Аналогично, почти все структуры данных являются производными от нескольких базовых типов. Поэтому материал,

рассматриваемый в этой главе, знаком практически всем программистам. Мы написали рабочие версии кода для того, чтобы иметь конкретный материал для обсуждения, и читатель может использовать соответствующие фрагменты кода практически дословно. Однако желательно делать это только после тщательного изучения ассортимента средств, предлагаемого тем или иным языком и его библиотеками.

2.1. Поиск

Если говорить о хранении статических табличных данных, то ничто не может сравниться по эффективности с массивом. Инициализация на этапе компиляции позволяет конструировать массивы самыми простыми средствами и без больших затрат ресурсов. (В языке Java инициализация происходит на этапе выполнения программы, но эти подробности реализации несущественны, если массив не слишком велик.) Пусть, например, мы пишем программу, подсчитывающую в тексте "слова-паразиты". Такие слова обычно встречаются слишком часто в плохо написанном тексте.

```
char *flab[] = {
    "actually",
    "just",
    "quite",
    "really",
    NULL
};
```

Подпрограмма поиска должна знать, сколько элементов содержится в массиве. Один из способов дать ей это понять заключается в том, чтобы передать длину в качестве аргумента, а другой способ (используемый здесь) состоит в том, чтобы поместить маркер `NULL` в конец массива:

```
/* lookup: последовательный поиск слова в массиве */
int lookup(char *word, char *array[])
{
    int i;
    for (i = 0; array[i] != NULL; i++)
        if (strcmp(word, array[i]) == 0)
            return i;
    return -1;
}
```

В языках C и C++ параметр, представляющий собой массив строк, можно объявить как `char *array[]` или `char **array`. Хотя эти формы эквивалентны, первый способ яснее указывает на способ использования объекта данных.

Данный алгоритм поиска называется *последовательным поиском*, потому что в нем перебираются все элементы подряд, пока не будет найден нужный. Если объем данных сравнительно невелик, последовательный поиск работает достаточно быстро. Существуют стандартные библиотечные функции для выполнения последовательного поиска в совокупностях данных различных типов. Например, такие функции, как `strchr` и `strstr`, разыскивают первое вхождение соответственно символа и подстроки в строке C/C++; класс `String` языка Java содержит метод `indexOf`

для той же цели; нетипизированные алгоритмы find языка C++ применимы к большинству типов данных. Если для типа данных, с которым вы работаете, уже есть стандартная функция, воспользуйтесь ею.

Последовательный поиск легко реализуется, но объем выполняемых в нем операций прямо пропорционален объему имеющихся данных. Удвоение количества элементов в массиве приведет к удвоению времени на поиск элемента, особенно если его нет среди данных. Это линейная зависимость, т.е. время поиска является линейной функцией объема данных; вот почему этот метод еще называется *линейным поиском*.

Ниже приведен фрагмент массива более реалистичного размера, взятый из программы синтаксического анализа языка HTML. В этом массиве определяются символические имена для более чем ста текстовых символов:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};
/* Символы HTML; например, Aelig обозначает лигатуру A и E. */
/* Значения соответствуют кодировке Unicode/ISO10646. */
Nameval htmlchars[] = {
    "AElig",    0x00c6,
    "Aacute",   0x00c1,
    "Acirc",    0x00c2,
    /* ... */
    "zeta", 0x03b6,
};
```

Для таких больших массивов, как этот, удобнее выполнять *поиск делением пополам* (*дихотомией*), или *двоичный поиск*. Алгоритм двоичного поиска — это математическая формализация того способа, которым мы, люди, разыскиваем слово в словаре. Сначала берем средний элемент. Если он больше того, который мы ищем, переходим к поиску в первой половине массива; в противном случае ищем во второй половине. Повторяем процедуру до тех пор, пока не найдем нужный элемент или не удостоверимся, что его нет среди имеющихся данных.

Для выполнения двоичного поиска таблица данных должна быть отсортирована. В данном случае это так; да и вообще, данные полезно сортировать — это и признак хорошего стиля, и помощь в поиске нужной информации. Кроме того, необходимо знать длину таблицы. В этом может помочь макрос NELEMS, приведенный в главе 1:

```
printf("The HTML table has %d words\n", NELEMS(htmlchars));
```

Функция двоичного поиска в этой таблице может выглядеть, например, таким образом:

```
    /* lookup: двоичный поиск строки name в массиве tab; возвращает
индекс */
    int lookup(char *name, Nameval tab[], int ntab)
    {
        int low, high, mid, cmp;

        low = 0;
        high = ntab - 1;
        while (low <= high) {
```

```
            mid = (low + high) / 2;
            cmp = strcmp(name, tab[mid].name);
            if (cmp < 0)
                high = mid - 1;
            else if (cmp > 0)
                low = mid + 1;
            else    /* элемент найден */
                return mid;
        }
        return -1;  /* элемент не найден */
    }
```

Подведем итог. Чтобы найти в массиве `htmlchars` индекс символа ½ ("одна вторая"), необходимо записать такой оператор:

```
    half = lookup("frac12", htmlchars, NELEMS(htmlchars));
```

В ходе двоичного поиска на каждом шаге отбрасывается половина оставшихся данных. Таким образом, процедура включает столько шагов, сколько раз можно поделить количество элементов на 2, пока не останется один элемент. Это равно $\log_2 n$, не считая ошибки округления. Если, например, массив содержит 1000 элементов, то линейный поиск может занять до 1000 шагов, тогда как двоичный — около 10. В массиве из миллиона элементов линейный поиск требует порядка миллиона операций сравнения, а двоичный — порядка 20. Чем больше в массиве элементов, тем большее преимущество имеет алгоритм двоичного поиска над линейным. Начиная с некоторого объема исходных данных (зависящего от конкретной реализации), двоичный поиск всегда работает быстрее, чем линейный.

2.2. Сортировка

Алгоритм двоичного поиска работает только в том случае, если элементы массива отсортированы. Если в некотором наборе данных предстоит часто выполнять поиск, то имеет смысл отсортировать его один раз, а затем использовать процедуру двоичного поиска для нахождения нужных элементов. Если набор данных известен заранее, его можно отсортировать на этапе написания программы и окончательно сформировать при компиляции. Если же это не так, то его придется сортировать на этапе выполнения программы.

Одним из лучших универсальных методов сортировки является алгоритм *быстрой сортировки* (*quicksort*), изобретенный в 1960 году Ч.Э.Р. Хоаром (C.A.R. Hoare). Этот алгоритм — превосходная демонстрация того, как можно обойтись без лишних вычислений. Для его реализации массив делится на "большие" и "малые" элементы:

выбирается один элемент в массиве (назовем его *опорным элементом* — *pivot*);
все остальные элементы делятся на две группы:
  "малые", т.е. меньшие, чем опорный;
  "большие", т.е. превосходящие по значению опорный или равные ему;
каждая из групп подвергается рекурсивной сортировке.

По окончании этого процесса массив оказывается упорядоченным. Этот алгоритм работает очень быстро, поскольку если известно, что какой-либо элемент меньше опорного, то уже нет нужды сравнивать его со всеми “большими” элементами. Аналогично, “большие” элементы не сравниваются с “малыми”. Поэтому данный алгоритм значительно быстрее, чем более простые наподобие сортировки простыми вставками или методом пузырьков, в которых каждый элемент непосредственно сравнивается со всеми остальными.

Алгоритм быстрой сортировки весьма практичен и эффективен; проведено множество исследований его свойств и разработано огромное количество его вариантов. Версия, которую мы приводим здесь, — одна из простейших, но далеко не самых быстрых.

Приведенная ниже функция `quicksort` сортирует массив целых чисел.

```
/* quicksort: сортирует v[0]...v[n-1] в порядке возрастания */
void quicksort(int v[], int n)
{
    int i, last;

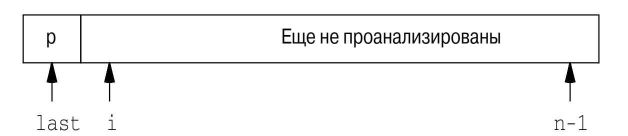
    if (n <= 1)  /* ничего не нужно делать */
        return;
    swap(v, 0, rand() % n);  /* переместить опору в v[0] */
    last = 0;
    for (i = 1; i < n; i++)  /* разбиение */
        if (v[i] < v[0])
            swap(v, ++last, i);
    swap(v, 0, last);        /* восстановить опору */
    quicksort(v, last);      /* рекурсивная сортировка */
     quicksort(v+last+1, n-last-1); /* каждой из частей */
}
```

Операция `swap`, меняющая местами два элемента, фигурирует в функции `quicksort` три раза, поэтому ее лучше всего оформить в виде отдельной функции:

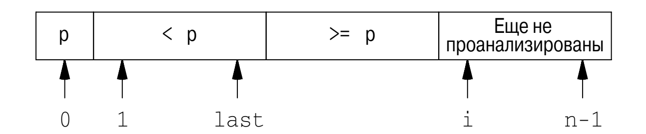
```
/* swap: меняет местами элементы v[i] и v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = v[i];
}
```

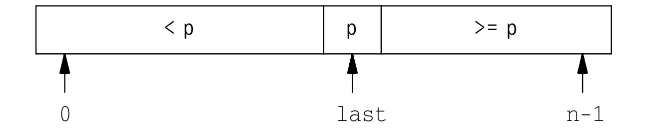
При разбиении массива в нем случайным образом выбирается опорный элемент; затем он временно перебрасывается в начало массива; выполняется перебор остальных элементов, причем “малые” (меньшие, чем опорный) перемещаются в начало (в позицию `last`), а “большие” — в конец (в позицию `i`). В начале этой процедуры, сразу после перемещения опорного элемента в начало, имеет место равенство `last = 0`, а элементы от `i = 1` до `n-1` еще не проанализированы:



Что касается перебора в цикле `for`, то элементы от `1` до `last` строго меньше, чем опорный, элементы от `last+1` до `i-1` больше или равны опорному, а элементы от `i` до `n-1` еще не проанализированы. До тех пор, пока не начнет выполняться соотношение `v[i] >= v[0]`, элемент `v[i]` может участвовать в обмене с самим собой; из-за этого тратится некоторое время, но не так уж много, чтобы об этом беспокоиться.



После разбиения всех элементов на группы элемент `0` меняется местами с элементом `last`, чтобы поместить опорный элемент в его окончательную позицию. Это позволяет сохранить правильный порядок. Теперь массив выглядит следующим образом:



Эта же процедура применяется к левому и правому подмассивам; как только она закончена, массив оказывается упорядоченным.

Насколько быстр алгоритм быстрой сортировки? В наилучшем возможном случае:

на первом этапе n элементов разбиваются на две группы примерно по $n/2$ элементов в каждой;

на втором этапе две группы по примерно $n/2$ элементов разбиваются на четыре по примерно $n/4$ элементов;

далее четыре группы разбиваются на восемь по $n/8$ элементов и т.д.

Процесс протекает примерно в $\log_2 n$ этапов, так что общий объем операций в наилучшем случае пропорционален $n + 2 \times n/2 + 4 \times n/4 + 8 \times n/8 ...$ ($\log_2 n$ слагаемых), что составляет $n \log_2 n$. В среднестатистическом случае эта процедура лишь немногим более трудоемка. Логарифм по основанию 2 — это стандарт, поэтому основание можно не писать и считать, что затраты времени на быструю сортировку пропорциональны $n \log n$.

Данная реализация быстрой сортировки наиболее удобна для демонстрации, но в ней есть и слабые места. Если при каждом выборе опорного элемента элементы разбиваются на две почти одинаковых по численности группы, то выполненный нами анализ правилен; однако если разбиение на группы часто оказывается слишком неравномерным, то время работы растет примерно по закону n^2. В нашей реализации опорный элемент выбирается случайно. Это снижает вероятность того, что необычность входных данных приведет к слишком большому количеству неравномерных разбиений. Но если все исходные данные одинаковы, то в нашем варианте каждый раз от группы будет отделяться всего один элемент, и время работы приобретет порядок n^2.

Эффективность работы некоторых алгоритмов сильно зависит от исходных данных. Случайно или преднамеренно искаженный характер данных может привести к тому, что вполне стабильный алгоритм вдруг станет работать очень медленно или использовать слишком много памяти. В случае быстрой сортировки наша простейшая реализация иногда работает слишком медленно, однако более совершенные варианты сводят вероятность аномального поведения алгоритма почти к нулю.

2.3. Библиотечные средства

Стандартные библиотеки С и С++ содержат функции сортировки, которые ведут себя устойчиво по отношению к любым входным данным и развивают максимально возможное быстродействие.

Библиотечные функции предназначены для сортировки данных любого типа, однако взамен пользователь должен приспособиться к их интерфейсу, который часто бывает несколько сложнее, чем рассмотренный ранее. В языке С библиотечная функция называется `qsort`, и для ее работы необходимо подготовить функцию сравнения, которая будет вызываться из `qsort` всякий раз, когда необходимо сравнить два значения. Поскольку значения могут иметь любой тип, в функцию сравнения передаются два указателя типа `void*` на сравниваемые элементы данных. Функция приводит указатели к требуемому типу, извлекает значения данных, сравнивает их и возвращает результат (отрицательный, нулевой или положительный, в зависимости от того, является ли первое значение бóльшим, равным или меньшим, чем второе).

Ниже показан вариант, предназначенный для сортировки массива строк (это часто встречающаяся задача). Определим функцию `scmp`, которая приводит аргументы к нужному типу и вызывает функцию `strcmp` для непосредственного сравнения:

```
/* scmp: сравнение строк *p1 и *p2 */
int scmp(const void *p1, const void *p2)
{
    char *v1, *v2;

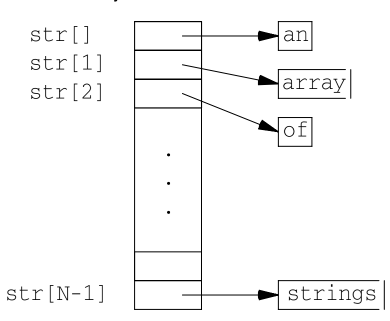
    v1 = *(char **) p1;
    v2 = *(char **) p2;
    return strcmp(v1, v2);
}
```

Все это можно было бы записать в одну строчку, но код читать удобнее с применением временных переменных.

Чтобы отсортировать элементы от `str[0]` до `str[N-1]` в массиве строк, в функцию `qsort` следует передать такие аргументы, как сам массив, его длина, размер сортируемых элементов и указатель на функцию сравнения:

```
char *str[N];
qsort(str, N, sizeof(str[0]), cmp);
```

Функцию `strcmp` нельзя использовать непосредственно как функцию сравнения, потому что `qsort` перебирает адреса всех элементов в массиве в виде `&str[i]` типа `char**`, а не `str[i]` типа `char*`, как показано на этом рисунке:



А вот аналогичная функция `icmp` для сравнения целых чисел:

```
/* icmp: сравнение целых чисел *p1 и *p2 */
int icmp(const void *p1, const void *p2)
{
    int v1, v2;

    v1 = *(int *) p1;
    v2 = *(int *) p2;
    if (v1 < v2)
        return -1;
    else if (v1 == v2)
        return 0;
    else
        return 1;
}
```

Можно было бы написать:

```
?   return v1-v2;
```

Но в том случае, если `v2` велико и положительно, а `v1` — велико и отрицательно (или наоборот), получившееся арифметическое переполнение приведет к неправильному ответу. Непосредственное сравнение длиннее, но безопаснее.

И в этом случае вызов функции `qsort` требует передачи массива, его длины, размера элемента и указателя на функцию сравнения в качестве аргументов:

```
int arr[N];

qsort(arr, N, sizeof(arr[0]), icmp);
```

В ANSI C также определена функция двоичного поиска под именем `bsearch`. Как и `qsort`, функции `bsearch` требуется указатель на функцию сравнения (часто это та же самая, что используется в `qsort`). Она возвращает указатель на найденный элемент или `NULL`, если элемент отсутствует. Далее приведена функция `lookup` для поиска символов HTML в кодовой таблице, переписанная с использованием процедуры `bsearch`.

```
/* lookup: ищет строку name в массиве tab, возвращает индекс */
int lookup(char *name, Nameval tab[], int ntab)
{
    Nameval key, *np;

    key.name = name;
    key.value = 0;  /* не используется; может быть любым */
    np = (Nameval *) bsearch(&key, tab, ntab,
                              sizeof(tab[0]), nvcmp);
    if (np == NULL)
        return -1;
    else
        return np-tab;
}
```

Как и в случае `qsort`, функция сравнения получает адреса сравниваемых элементов, поэтому ключ `key` должен иметь соответствующий тип. В данном примере приходится конструировать фиктивную запись `Nameval`, чтобы было что передать в функцию сравнения. Сама функция носит имя `nvcmp` и сравнивает два элемента `Nameval` путем вызова стандартной функции `strcmp` с ее строковыми компонентами в качестве аргументов. Числовые значения игнорируются.

```
/* nvcmp: сравнивает два имени типа Nameval */
int nvcmp(const void *va, const void *vb)
{
    const Nameval *a, *b;
    a = (Nameval *) va;
    b = (Nameval *) vb;
    return strcmp(a->name, b->name);
}
```

Эта функция аналогична `scmp`, но отличается тем, что строки представлены в виде полей структур.

Неудобство, связанное с использованием ключа, показывает, что функция `bsearch` предоставляет пользователю меньшие функциональные возможности, чем `qsort`. Хорошая подпрограмма сортировки общего назначения занимает несколько страниц кода, тогда как функция двоичного поиска обычно не длиннее, чем те несколько операторов, которые обеспечивают интерфейс к ней в программе. И тем не менее рекомендуется пользоваться `bsearch`, а не писать свою собственную функцию. Уже в течение многих лет задача двоичного поиска является на удивление трудной для большинства программистов.

Стандартная библиотека C++ содержит нетипизированный алгоритм под названием `sort`, который гарантирует количество операций порядка $O(n \log n)$. Его интерфейс проще, чем у рассмотренных функций, потому что он не требует ни приведения типов, ни знания размера элементов, ни явно заданной функции сравнения, лишь бы тип исходных данных предполагал отношение упорядоченности.

```
int arr[N];

sort(arr, arr+N);
```

В библиотеке C++ также имеются нетипизированные процедуры двоичного поиска с теми же преимуществами интерфейса.

Упражнение 2.1. Самая естественная форма алгоритма быстрой сортировки — это рекурсивная. Напишите этот же алгоритм в итерационной форме и сравните две его версии. (Ч. Хоар рассказывал, как тяжело было реализовать быструю сортировку итерационным способом и как все стало на свои места, стоило только перейти к рекурсивной форме.)

2.4. Быстрая сортировка в Java

Ситуация в языке Java несколько отличается от C. В ранних версиях Java не было стандартной функции сортировки, и приходилось писать собственные. А вот в более новых версиях функция `sort` уже есть, и она работает с классами, реализующими интерфейс `Comparable`, так что теперь можно сортировать данные библиотечными средствами. Но поскольку сама техника сортировки может пригодиться и в других приложениях, далее мы проработаем все детали реализации алгоритма быстрой сортировки на языке Java.

Нетрудно адаптировать функцию `quicksort` к данным любых типов, сортировка которых может нам понадобиться, но будет более поучительно реализовать процедуру нетипизированной сортировки, которую можно вызывать для любого объекта, т.е. в стиле интерфейса `qsort`.

Существенное отличие этого случая от C и C++ состоит в том, что в языке Java невозможно передать функцию сравнения в другую функцию: там нет указателей на функции. Вместо этого создается *интерфейс*, содержащий одну функцию, которая сравнивает два объекта типа `Object`. Затем для каждого класса сортируемых данных создается класс с методом, реализующим интерфейс для этого конкретного типа. Экземпляр этого класса передается в функцию сортировки, которая в свою очередь пользуется функцией сравнения из состава класса для того, чтобы сравнивать сортируемые элементы.

Начнем с определения интерфейса `Cmp`, в котором объявляется один член — функция сравнения `cmp`, сравнивающая между собой два объекта `Object`.

```
interface Cmp {
    int cmp(Object x, Object y);
}
```

Теперь можно написать функции сравнения для реализации этого интерфейса. Например, в следующем классе определяется функция для сравнения двух объектов типа `Integer`:

```
// Icmp: сравнение объектов Integer
class Icmp implements Cmp {
    public int cmp(Object o1, Object o2);
    {
        int i1 = ((Integer) o1).intValue();
        int i2 = ((Integer) o2).intValue();
        if (i1 < i2)
            return -1;
```

```
            else if (i1 == i2)
                return 0;
            else
                return 1;
        }
    }
```

В этом классе сравнению подвергаются объекты `String`:

```
    // Scmp: String comparison
    class Scmp implements Cmp {
        public int cmp(Object o1, Object o2)
        {
            String s1 = (String) o1;
            String s2 = (String) o2;
            return s1.compareTo(s2);
        }
    }
```

Таким способом можно сортировать только данные типов, производных от `Object`; элементы базовых типов наподобие `int` или `double` подобным образом отсортировать нельзя. Вот почему в нашем примере мы сортируем объекты класса `Integer`, а не типа `int`.

Имея эти компоненты, уже можно перевести функцию `quicksort` с языка C на язык Java и сделать так, чтобы она вызывала функцию сравнения из объекта `Cmp`, переданного в нее как аргумент. Наиболее существенное изменение состоит во введении индексов `left` и `right`, поскольку в Java нет указателей на массивы.

```
    // Quicksort.sort: быстрая сортировка элементов v[left]..v[right]
    static void sort(Object[] v, int left, int right, Cmp cmp)
    {
        int i, last;

        if (left >= right ) // ничего не нужно делать
            return;
        swap(v, left, rand(left, right)));  // переместить опорный элемент
        last = left;                        // в v[left]
        for (i = left+1; i <= right; i++)   // разбиение диапазона
            if (cmp.cmp(v[i], v[left]) < 0)
                swap(v, ++last, i);
        swap(v, left, last);                // восстановить опорный элемент
        sort(v, left, last-1, cmp);         // рекурсивная сортировка
        sort(v, last+1, right, cmp);        // обеих частей диапазона
    }
```

Функция `Quicksort.sort` использует функцию `cmp` для сравнения пары объектов между собой, а затем вызывает `swap`, как и раньше, чтобы поменять их местами.

```
    // Quicksort.swap: поменять местами v[i] и v[j]
    static void swap(Object[] v, int i, int j)
    {
        Object temp;

        temp = v[i];
        v[i] = v[j];
```

```
        v[j] = temp;
    }
```

Генерирование случайных чисел осуществляется отдельной функцией в диапазоне от `left` до `right` включительно:

```
    static Random rgen = new Random();

    // Quicksort.rand: возвращает случайное целое число
    //                 в диапазоне [left,right]
    static int rand(int left, int right)
    {
        return left + Math.abs(rgen.nextInt())%(right-left+1);
    }
```

Здесь с помощью функции `Math.abs` вычисляется абсолютное значение, потому что генератор случайных чисел Java выдает и положительные, и отрицательные числа.

Функции `sort`, `swap` и `rand`, а также объект-генератор `rgen` являются членами класса `Quicksort`.

Наконец, чтобы воспользоваться функцией `Quicksort.sort` для сортировки массива данных типа `String`, запишем следующее:

```
    String[] sarr = new String[n];

    // Здесь заполняется n элементов массива sarr...

    Quicksort.sort(sarr, 0, sarr.length-1, new Scmp());
```

В результате этого функция `sort` вызывается с аргументом в виде объекта для сравнения строк, созданного специально для данного случая.

Упражнение 2.2. Наша реализация быстрой сортировки на Java изобилует операциями приведения типов, поскольку элементы постоянно преобразовываются из их исходного типа (например, `Integer`) в тип `Object` и наоборот. Напишите версию `Quicksort.sort` для сортировки данных вполне определенного типа и поэкспериментируйте с обоими вариантами функции, чтобы выяснить, какое влияние на быстродействие оказывают преобразования типов.

2.5. *O*-оценка

Ранее объем операций, выполняемый тем или иным алгоритмом, уже характеризовался через n — количество элементов в наборе исходных данных. Поиск в несортированных данных требует времени, пропорционального n; если воспользоваться алгоритмом двоичного поиска в отсортированных данных, то затрачиваемое время окажется пропорциональным $\log n$. Время сортировки данных может измеряться порядком n^2 или $n \log n$.

Нам необходим способ формулировать подобные зависимости более четко и конкретно, в то же время абстрагируясь от таких деталей, как быстродействие процессора или качество компилятора (а также квалификация программиста). Желательно было бы сравнивать время выполнения и требуемые объемы памяти тех или иных алгоритмов независимо от языка программирования, компилятора, аппаратной

архитектуры, быстродействия процессора, загрузки операционной системы и других усложняющих анализ факторов.

Для этой цели имеется стандартная форма записи или метод оценки, именуемый O-оценкой. Основным ее параметром является n, т.е. размер исследуемой задачи, причем время выполнения алгоритма или его *сложность* представляется в виде функции n. Буква O представляет собой сокращение от слова *order* ("порядок"). Таким образом, выражение "двоичный поиск имеет характер $O(\log n)$" означает, что требуется порядка $\log n$ шагов для поиска в массиве из n элементов. Запись $O(f(n))$ означает, что при возрастании n время выполнения алгоритма растет пропорционально $f(n)$, например $O(n^2)$ или $O(n \log n)$. Подобные асимптотические оценки играют большую роль в теоретическом анализе, а также очень помогают при грубом сравнении алгоритмов, но на практике возникают значительные отличия в деталях. Например, алгоритм порядка $O(n^2)$ с небольшим коэффициентом пропорциональности может работать быстрее, чем алгоритм порядка $O(n \log n)$ с высоким коэффициентом для относительно малых n, но по мере роста n преимущество будет оставаться за алгоритмом с медленнее растущей функцией в O-оценке.

Следует также различать поведение алгоритмов в *наихудшем* и *среднестатистическом* случаях. Что такое "среднестатистический случай", определить довольно трудно, поскольку это зависит от предполагаемого характера входных данных. Наихудший случай обычно можно определить более-менее точно, хотя он легко может дезориентировать программиста. Например, для алгоритма быстрой сортировки наихудший случай соответствует количеству операций порядка $O(n^2)$, а его среднее быстродействие характеризуется оценкой $O(n \log n)$. Всякий раз тщательно выбирая опорный элемент, можно свести вероятность квадратичного, т.е. $O(n^2)$, объема операций практически к нулю. На практике хорошо написанная процедура быстрой сортировки почти всегда выполняет порядка $O(n \log n)$ операций.

Ниже приведены наиболее важные на практике порядковые оценки.

Оценка	*Характер зависимости*	*Пример операции*
$O(1)$	Постоянная	Обращение по индексу
$O(\log n)$	Логарифмическая	Двоичный поиск
$O(n)$	Линейная	Сравнение строк
$O(n \log n)$	Пропорциональная $n \log n$	Быстрая сортировка
$O(n^2)$	Квадратичная	Простая сортировка
$O(n^3)$	Кубическая	Умножение матриц
$O(2^n)$	Экспоненциальная	Разбиение на группы

Обращение к элементу в массиве имеет порядок $O(1)$, т.е. не зависит от количества элементов. Алгоритм, который на каждом шаге отбрасывает половину оставшихся данных, наподобие двоичного поиска, обычно требует $O(\log n)$ операций. Сравнение двух строк из n элементов с помощью функции `strcmp` имеет характер $O(n)$. Традиционный алгоритм умножения матриц отнимает $O(n^3)$ времени, поскольку каждый элемент на выходе образуется путем перемножения n пар исходных элементов и сложения результатов, а всего в матрице n^2 элементов.

Алгоритмы с экспоненциальным порядком количества операций часто включают в себя вычисление или перебор всех возможных сочетаний и вариантов. Так, в множестве из n элементов имеется 2^n подмножеств, поэтому если в алгоритме выполняется перебор по всем подмножествам, то он имеет характер $O(2^n)$. Экспоненциальные алгоритмы обычно обходятся слишком дорого, если n достаточно велико, потому что добавление в задачу всего одного элемента удваивает количество операций. К сожалению, существует немало задач, в том числе знаменитая "задача коммивояжера", для решения которых известны только алгоритмы экспоненциального порядка. Если необходимо решить именно такую задачу, очень часто прибегают к альтернативным методам, которые дают то или иное приближение к точному правильному ответу.

Упражнение 2.3. Выясните, какие исходные данные являются наихудшими для алгоритма быстрой сортировки. Попытайтесь найти несколько массивов данных, которые бы заставили библиотечную реализацию этого алгоритма работать как можно медленнее. Автоматизируйте процесс экспериментирования так, чтобы подбор и анализ исходных данных выполнялись сами по себе.

Упражнение 2.4. Разработайте и реализуйте алгоритм, сортирующий массив из n целых чисел как можно медленнее. Играть нужно честно: алгоритм должен постепенно сходиться к результату и успешно заканчиваться; нельзя применять такие обманные трюки, как холостые прогоны циклов специально для задержки времени. Какой порядок имеет этот алгоритм по отношению к количеству элементов n?

2.6. Расширяемые массивы

В предыдущих разделах использовались статические массивы, т.е. такие, размер и содержимое которых фиксировались еще на этапе компиляции. Если бы таблицы "слов-паразитов" или символов HTML приходилось модифицировать в процессе выполнения программы, то в качестве базовой структуры данных больше подошла бы хэш-таблица. Добавление в отсортированный массив n элементов по одному за раз является операцией порядка $O(n^2)$, и ее лучше избегать при больших n.

Тем не менее иногда бывает необходимо держать все данные непостоянного объема в одной составной переменной, причем этих данных не должно быть слишком много. Самой удачной структурой данных для подобной задачи по-прежнему остается массив. Чтобы минимизировать затраты на распределение памяти, приращение массива следует выполнять блоками, и для поддержания хорошего стиля программирования массив нужно хранить вместе со статистической информацией, необходимой для его обработки. В языках C++ и Java это делается с помощью классов из стандартных библиотек, а в C — с помощью структур.

В приведенном ниже фрагменте определяется растущий массив элементов типа `Nameval`; новые элементы добавляются в конец массива, который расширяется по мере необходимости. Имеется возможность обратиться к любому элементу массива по его индексу за фиксированное время. Получается некий аналог векторных классов из библиотек Java и C++.

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
};

struct Nvtab {
    int     nval;         /* текущее количество элементов */
    int     max;          /* количество выделенных ячеек */
    Nameval *nameval;     /* массив пар "имя-значение" */
}

enum ( NVINIT = 1, NVGROW = 2 };

/* addname: добавляет новое имя и значение в структуру nvtab */
int addname(Nameval newname)
{
    Nameval *nvp;

    if(nvtab.nameval == NULL) { /* первый раз */
       nvtab.nameval =
           (Nameval *) malloc(NVINIT * sizeof(Nameval));
       if (nvtab.nameval == NULL)
           return -1;
       nvtab.max = NVINIT;
       nvtab.nval = 0;
    } else if (nvtab.nval >= nvtab.max) { /* расширение */
       nvp = (Nameval *) realloc(nvtab.nameval,
               (NVGROW*nvtab.max) * sizeof(Nameval));
       if (nvp == NULL)
           return -1;
       nvtab.max *= NVGROW;
       nvtab.nameval = nvp;
    }
    nvtab.nameval[nvtab.nval] = newname;
    return nvtab.nval++;
}
```

Функция `addname` возвращает индекс только что добавленного элемента или `-1`, если произошла ошибка.

При вызове функции `realloc` массив расширяется до нового размера, сохраняя все уже имеющиеся в нем элементы. Возвращается указатель на новый буфер или `NULL`, если оказалось недостаточно памяти. Удвоение размера массива при каждом вызове `realloc` позволяет сохранить постоянным средний объем операций по копированию одного элемента. Если бы при каждом вызове этой функции добавлялась всего одна ячейка, время таких операций имело бы порядок $O(n^2)$. Поскольку адрес массива может изменяться при перераспределении памяти, в остальной части программы следует обращаться к элементам по индексам, а не через указатели. Обратите внимание, что в коде не употребляется такая конструкция:

```
?   nvtab.nameval = (Nameval *) realloc(nvtab.nameval,
?           (NVGROW*nvtab.max) * sizeof(Nameval));
```

В этом случае, если бы при перераспределении памяти произошла ошибка, все накопленные в исходном массиве данные были бы потеряны.

Работа начинается с массива очень малого исходного размера (NVINIT = 1). Это заставляет программу немедленно расширить его и таким образом гарантирует, что данный блок программы сработает. Первоначальный размер массива можно и увеличить перед тем, как выпускать программу для профессионального использования, но, вообще-то, затраты на инициализацию не так уж велики, чтобы об этом беспокоиться.

Возвращаемое из realloc значение не обязательно приводить к его окончательному типу, потому что в языке C это делается с указателями *void автоматически. А вот в C++ это не так; там явное приведение типа необходимо. Можно поспорить о том, что безопаснее: приводить типы (это более явная запись, в хорошем стиле) или не приводить (приведение типов может скрыть различные тонкие ошибки). Мы решили включить в функцию приведение типов, чтобы сделать код безошибочным с точки зрения как C, так и C++. В результате компилятор C менее тщательно проверяет ошибки, но это уравновешивается дополнительной возможностью проверки сразу двумя компиляторами.

Удаление имени из массива может оказаться довольно хитрой задачей, поскольку нужно решить, что делать с освободившейся ячейкой. Если порядок элементов не имеет значения, то легче всего перебросить последний элемент в образовавшуюся свободную позицию. Если же порядок необходимо соблюсти, то придется сдвинуть на одну ячейку назад все элементы после свободной позиции:

```
    /* delname: удаление первой найденной строки nameval из массива
nvtab */
    int delname(char *name)
    {
        int i;

        for (i = 0; i < nvtab.nval; i++)
            if (strcmp(nvtab.nameval[i].name, name) == 0) {
                memmove(nvtab.nameval+i, nvtab.nameval+i+1,
                    (nvtab.nval-(i+1)) * sizeof(Nameval));
                nvtab.nval--;
                return 1;
            }
        return 0;
    }
```

При вызове функции memmove массив “сдавливается”, сдвигаясь внутрь себя на одну позицию. Эта функция принадлежит к стандартной библиотеке и предназначена для копирования блоков памяти произвольной длины.

В стандарте ANSI C определены две функции: memcpy, которая работает быстро, но затирает память, если буфер-источник и буфер назначения перекрываются; и memmove, работающая медленнее, зато неизменно правильно. Бремя выбора между правильностью и быстродействием не следовало бы возлагать на программиста, так что в идеале функция должна быть только одна. Представим себе, что так оно и есть, и всегда будем пользоваться memmove.

Вызов функции memmove можно было бы заменить следующим циклом:

```
int j;
for (j = i; j < nvtab.nval-1; j++)
    nval.nameval[j] = nvtab.nameval[j+1];
```

Мы предпочитаем memmove, потому что ее использование позволяет избежать характерной ошибки копирования элементов в неправильном порядке. Если бы выполнялась вставка, а не удаление, элементы было бы необходимо перебирать в обратном порядке, чтобы не затереть их. Вызывая memmove, мы избавляемся от необходимости всякий раз думать об этом.

Подход, альтернативный перемещению элементов в массиве, — это пометить удаленные элементы как неиспользуемые. Затем, чтобы добавить новый элемент, вначале необходимо найти свободную ячейку, и только в том случае, если таковых нет, расширить массив до нового размера. В данном примере, чтобы пометить элемент как неиспользуемый, в его поле name можно записать значение NULL.

Массивы — это простейший способ группировки данных. Неудивительно, что во многих языках имеются эффективные и удобные средства для их хранения и индексации и что строки представляются именно массивами символов. Массивы просты в использовании, требуют $O(1)$ операций для обращения к любым элементам, отлично подходят для применения двоичного поиска и быстрой сортировки, компактно хранятся в памяти. Для целей хранения наборов данных фиксированного размера, формируемых в процессе компиляции, или относительно небольших совокупностей элементов переменной длины массивы не знают себе равных. Но работа с постоянно меняющимся набором элементов в массиве переменной длины может отнять много времени и ресурсов, поэтому если количество элементов непредсказуемо и может стать очень большим, то лучше обратиться к другим структурам данных.

Упражнение 2.5. В приведенном выше коде функция delname не вызывает функцию realloc, чтобы возвратить память, освободившуюся при удалении. Стоит ли это делать? Как бы вы определили, делать это или нет?

Упражнение 2.6. Внесите необходимые изменения в функции addname и delname так, чтобы удалять имена, помечая их как неиспользуемые. Насколько остальная часть программы изолирована от этих изменений?

2.7. Списки

Вслед за массивами самыми распространенными структурами данных в программах являются списки. Во многих языках есть встроенные списковые типы данных, а некоторые языки (такие как Lisp) вообще основаны на списках. Но в языке C приходится конструировать их самостоятельно. В C++ и Java имеются библиотечные реализации списков, но все равно нужно знать, когда и как ими пользоваться. В этом разделе рассматриваются списки в C, но эта информация может пригодиться и в более широком контексте.

Однонаправленный список — это набор элементов, каждый из которых состоит из хранимых данных и указателя на следующий элемент. Голова списка указывает на его первый элемент, а конец списка обозначается нулевым указателем. Здесь показан список из четырех элементов:



Между массивами и списками есть несколько важных различий. Во-первых, массив имеет фиксированную длину, тогда как список — в точности такую, какая нужна для хранения всех его данных, плюс некоторое дополнительное пространство для хранения указателей. Во-вторых, список можно переупорядочить, всего лишь поменяв местами несколько указателей, что гораздо удобнее, чем перемещать туда-сюда блоки данных. Наконец, при добавлении или удалении новых элементов остальные элементы никуда не перемещаются. Если, например, указатели на элементы списка хранятся в какой-то другой структуре данных, то ни один из них не потеряет свой смысл при добавлении новых элементов.

Все эти различия подсказывают, что если набор данных должен часто изменяться, в частности, если количество элементов заранее не предсказуемо, то такие данные следует организовать в виде списка. А вот массив больше подходит для сравнительно статичных данных.

Имеется всего лишь несколько базовых операций над списком: добавление нового элемента в начало или в конец списка; поиск элемента; добавление нового элемента перед определенным или после него; иногда удаление элемента. Простота устройства списков позволяет по необходимости добавлять дополнительные операции.

Вместо того чтобы определять отдельный тип `List`, в языке C обычно берут структурный тип данных для хранения элементов (например, рассмотренный выше `Nameval`) и добавляют указатель для связи со следующим элементом:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;  /* следующий в списке */
};
```

Непустой список трудно инициализировать на этапе компиляции, поэтому в отличие от массивов списки чаще создаются динамически. Вначале необходимо получить возможность конструировать элементы. Самый прямой способ — это выполнить распределение памяти соответствующей функцией, которую мы назовем `newitem`:

```
/* newitem: создает новый элемент по имени и значению */
Nameval *newitem(char *name, int value)
{
    Nameval *newp;

    newp = (Nameval *) emalloc(sizeof(Nameval));
```

```
        newp->name = name;
        newp->value = value;
        newp->next = NULL;
        return newp;
    }
```

Функция emalloc будет часто использоваться в этой книге. В ней вызывается malloc, а если при распределении памяти происходит ошибка, то выводится сообщение об ошибке и программа завершается. Ее код будет приведен в главе 4, а пока будем воспринимать ее как функцию распределения памяти, никогда не дающую ошибочного результата.

Самый простой и быстрый способ собрать список — это добавлять каждый новый элемент по очереди в голову списка:

```
    /* addfront: добавляет newp в голову списка listp */
    Nameval *addfront(Nameval *listp, Nameval *newp)
    {
        newp->next = listp;
        return newp;
    }
```

При модификации списка его первый элемент может измениться, как это и происходит в случае вызова функции addfront. Функции, модифицирующие список, должны возвращать указатель на его новый первый элемент, который хранится в переменной, обозначающей весь список в целом. Функция addfront и другие функции в этой группе возвращают указатель на первый элемент; типичный способ их вызова имеет вид

```
    nvlist = addfront(nvlist, newitem("smiley", 0x263A));
```

Эта конструкция работает даже в том случае, если существующий список — пустой (нулевой), и позволяет легко комбинировать функции в выражениях. Она удобнее, чем альтернативный способ с передачей указателя на указатель, содержащий голову списка.

Добавление элемента в конец списка — это операция порядка $O(n)$, поскольку список необходимо сначала перебрать от начала до конца:

```
    /* addend: добавляет newp в конец списка listp */
    Nameval *addend(Nameval *listp, Nameval *newp)
    {
        Nameval *p;

        if (listp == NULL)
            return newp;
        for (p = listp; p->next != NULL; p = p->next)
            ;
        p->next = newp;
        return listp;
    }
```

Если нужно придать функции addend характер $O(1)$ по быстродействию, можно завести отдельный указатель на конец списка. Недостаток этого подхода, кроме необходимости хранить и отслеживать хвостовой указатель, состоит в том, что спи-

сок больше нельзя будет представить одной-единственной адресной переменной. Поэтому будем придерживаться более простого стиля.

Для поиска элемента с тем или иным именем следует пройти по цепочке указателей `next`:

```
/* lookup: последовательный поиск имени name в списке listp */
Nameval *lookup(Nameval *listp, char *name)
{
    for ( ; listp != NULL; listp = listp->next)
        if (strcmp(name, listp->name) == 0)
            return listp;
    return NULL;    /* элемент не найден */
}
```

Эта операция занимает $O(n)$ времени, и в целом улучшить ее быстродействие не представляется возможным. Даже если список отсортирован, все равно необходимо перебрать его последовательно для нахождения нужного элемента. Двоичный поиск к спискам неприменим.

Для вывода всех элементов списка можно написать функцию, которая бы перебирала его и последовательно выводила каждый элемент. Чтобы вычислить длину списка, можно написать функцию с простым перебором и инкрементированием счетчика. Есть и альтернативный подход — написать функцию `apply`, выполняющую перебор списка и вызывающую другую заданную функцию для каждого его элемента. Функцию `apply` можно сделать очень гибкой, включив в ее параметры аргумент для передачи в ту, другую функцию. Итак, `apply` будет принимать три аргумента: сам список; указатель на функцию, вызываемую для каждого элемента списка; аргумент для передачи в эту функцию.

```
/* apply: выполняет fn для каждого элемента списка listp */
void apply(Nameval *listp,
        void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next)
        (*fn)(listp, arg);  /* вызов функции */
}
```

Второй аргумент функции `apply` — это указатель на функцию, которая принимает два аргумента и возвращает `void`. Она имеет довольно неуклюжий, хотя и стандартный синтаксис:

```
void (*fn)(Nameval*, void*)
```

Здесь `fn` объявляется как указатель на функцию, возвращающую пустое значение `void`, т.е. это переменная, содержащая адрес функции, которая ничего не возвращает. Функция принимает два аргумента. Один из них — это указатель на элемент списка `Nameval*`, а второй — нетипизированный указатель на аргумент для передачи в вызываемую функцию.

Чтобы использовать `apply`, например, для вывода элементов списка, можно написать тривиальную функцию с аргументом в виде строки формата:

```
/* printnv: вывести имя и значения по строке формата arg */
void printnv(Nameval *p, void *arg)
{
    char *fmt;

    fmt = (char *) arg;
    printf(fmt, p->name, p->value);
}
```

В данном случае функция `apply` вызывается таким образом:

```
apply(nvlist, printnv, "%s: %x\n");
```

Для подсчета элементов определяется функция с аргументом в виде указателя на целочисленный счетчик, который нужно инкрементировать:

```
/* inccounter: инкрементирует счетчик *arg */
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p не используется */
    ip = (int *) arg;
    (*ip)++;
}
```

Эта функция вызывается следующим образом:

```
int n;

n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

Не каждую операцию со списком можно реализовать в таком виде. Например, для уничтожения списка необходимо действовать более осторожно:

```
/* freeall: освобождение всех элементов списка listp */
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next) {
        next = listp->next;
        /* name освобождается в другом месте */
        free(listp);
    }
}
```

Содержимое памяти нельзя использовать после ее освобождения, поэтому `listp->next` необходимо сохранить в локальной переменной под именем `next` прежде, чем освобождать элемент, на который указывает переменная `listp`. Почему нельзя записать этот цикл, как и прежде, в следующем виде?

```
?    for ( ; listp != NULL; listp = next) {
?        free(listp);
```

В этом случае значение listp->next пропадет при вызове функции free, и произойдет ошибка.

Обратите внимание, что функция freeall не освобождает поле listp->name. Предполагается, что поле name каждого объекта Nameval освобождается где-то в другом месте или же что память для него вообще не отводилась. Чтобы распределение и освобождение памяти для элементов происходило по единой схеме, необходимо согласовать между собой функции newitem и freeall. Необходимо найти компромисс, гарантирующий освобождение памяти и в то же время сохранение тех элементов, которые нужно сохранить в памяти. В подобных случаях очень часто возникают ошибки. В других языках, таких как Java, эту проблему автоматически решает механизм сборки мусора. Мы еще вернемся к теме управления ресурсами в главе 4.

Удаление одного элемента из списка требует большего труда, чем добавление:

```
/* delitem: удаляет первое имя name из списка listp */
Nameval *delitem(Nameval *listp, char *name)
{
    Nameval *p, *prev;

    prev = NULL;
    for (p = listp; p != NULL; p = p->next) {
        if (strcmp(name, p->name) == 0) {
            if (prev == NULL)
                listp = p->next;
            else
                prev->next = p->next;
            free(p);
            return listp;
        }
        prev = p;
    }
    eprintf("delitem: %s not in list", name);
    return NULL;    /* сюда управление не доходит */
}
```

Как и freeall, функция delitem не освобождает поля name.

Функция eprintf выводит на экран сообщение об ошибке и завершает работу программы. Это неуклюже, если не сказать больше. Корректная обработка ошибок представляет собой довольно трудную задачу и требует более подробного обсуждения, которое мы отложим до главы 4. В главе 4 также будет продемонстрирована реализация функции eprintf.

Этих базовых структур и операций со списками вполне достаточно для реализации подавляющего большинства распространенных алгоритмов. Но есть и ряд альтернативных средств. Некоторые библиотеки, например стандартная библиотека шаблонов (STL) языка C++, поддерживают двунаправленные списки, в которых каждый узел содержит два указателя — на следующий и предыдущий элементы. В двунаправленных списках несколько больше дополнительные затраты памяти, но зато операции нахождения последнего элемента и удаления текущего имеют характер $O(1)$. В некоторых версиях указатели списков хранятся отдельно от данных, ко-

торые они связывают в список. Такие списки сложнее в обращении, но зато позволяют включать одни и те же элементы сразу в несколько разных списков.

Кроме того, что списки хорошо подходят для тех задач, где выполняется вставка и удаление элементов из середины структуры данных, они еще удобны для управления неупорядоченными данными варьирующегося объема, особенно если обращение к ним выполняется в стековом стиле — "последним вошел, первым вышел". Списки обеспечивают более эффективное использование памяти, чем массивы, в тех случаях, когда сразу несколько стеков должно расширяться и сокращаться независимо друг от друга. Удобно применять списки и тогда, когда данные упорядочены оригинальным способом, а их объем неизвестен априори, как это бывает в случае цепочки слов в текстовом документе. Но если приходится сочетать частое обновление данных с возможностью прямого доступа к ним, то лучше прибегнуть к структуре данных с менее ярко выраженной линейностью, например, к дереву или хэш-таблице.

Упражнение 2.7. Реализуйте некоторые из других возможных операций со списком: копирование, слияние, разбиение, вставку перед конкретным элементом или после него. Как две различные операции вставки отличаются друг от друга по сложности? В какой степени можно воспользоваться приведенными выше функциями, а что придется написать самостоятельно?

Упражнение 2.8. Напишите рекурсивную и итерационную версии функции `reverse`, которая бы изменяла порядок следования элементов в списке на противоположный. Не создавайте новых элементов списка — используйте только существующие.

Упражнение 2.9. Напишите нетипизированное определение типа `List` на языке C. Самый простой способ — это включить в состав каждого элемента списка указатель на данные типа `void*`. Проделайте то же самое на языке C++ в виде шаблона, а также на Java, определив класс для списка, содержащего элементы типа `Object`. Каковы сильные и слабые стороны разных языков в реализации этой задачи?

Упражнение 2.10. Придумайте и реализуйте набор тестов для проверки правильности написанных вами функций работы со списками. Стратегия и тактика тестирования рассматриваются в главе 6.

2.8. Деревья

Дерево — это еще одна иерархическая структура данных, содержащая набор элементов. Каждый элемент содержит значение определенного типа, может указывать на один, нуль или несколько других элементов, и при этом на него указывает только один другой элемент. Исключение составляет *корень* дерева — на этот элемент не указывает никакой другой.

Существует много типов деревьев, соответствующих сложным структурам данных из практических задач, — например, синтаксическое дерево, описывающее синтаксис предложения или оператора языка, или фамильное дерево, описывающее родственные отношения между людьми. Мы проиллюстрируем общие принципы на примере двоичных деревьев (точнее, деревьев двоичного поиска), которые в каждом узле имеют ровно две связи. Такие деревья легче всего реализовать, и при этом они

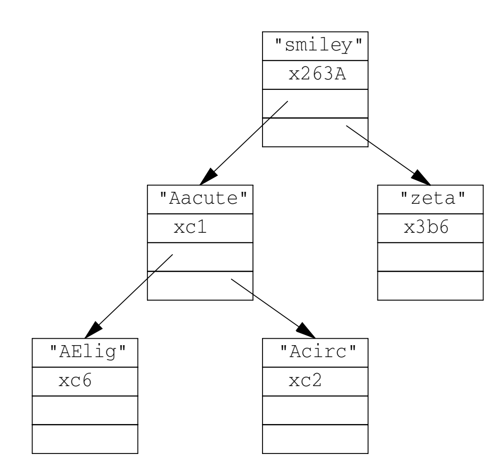
демонстрируют все существенные свойства этого класса структур данных. Каждый узел в двоичном дереве содержит значение и два указателя, `left` и `right`, указывающие соответственно на левый и правый дочерние узлы. Один или оба из этих указателей могут быть равны `NULL`, если узел имеет менее двух потомков. В дереве двоичного поиска значения в узлах полностью определяют структуру дерева: все потомки слева от какого-либо узла имеют меньшие значения, а все потомки справа от него — бо́льшие. В силу этого свойства в таком дереве можно использовать вариант алгоритма двоичного поиска для быстрого нахождения узла по значению.

Вариант структурного типа `Nameval` для использования в дереве строится очень легко:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *left;  /* меньшее значение */
    Nameval *right; /* большее значение */
};
```

Комментарии о меньшем и большем значениях соответствуют свойствам связей в таком дереве: в левых потомках узла хранятся меньшие значения, тогда как в правых — бо́льшие, чем в самом узле.

Ниже на рисунке приведен конкретный пример дерева — подмножество таблицы символов и их имен, организованное в виде двоичного дерева узлов типа `Nameval`, отсортированных по ASCII-кодам символов.



Поскольку в каждом узле имеются два указателя на другие элементы дерева, многие операции порядка $O(n)$ в списках или массивах имеют характер $O(\log n)$ в деревьях. Наличие двух указателей существенно уменьшает объем операций по перебору дерева, поскольку сужает круг узлов, перебираемых для нахождения элемента.

Дерево двоичного поиска (далее будем называть его просто “деревом”) строится таким образом: выполняется рекурсивный спуск с разветвлением влево или вправо, пока не найдется подходящее место для вставки нового узла. Узел должен представлять собой корректно инициализированный объект типа `Nameval` с именем, значением и двумя пустыми указателями. Новый узел добавляется в виде *листа* дерева, временно не имея потомков.

```
/* insert: вставляет newp в дерево treep, возвращает treep */
Nameval *insert(Nameval *treep, Nameval *newp)
{
    int cmp;

    if (treep == NULL)
        return newp;
    cmp = strcmp(newp->name, treep->name);
    if (cmp == 0)
        weprintf("insert: duplicate entry %s ignored",
            newp->name);
    else if (cmp < 0)
        treep->left = insert(treep->left, newp);
    else
        treep->right = insert(treep->right, newp);
    return treep;
}
```

Ранее мы ничего не сказали о дублирующихся записях в дереве. Данная версия функции `insert` сообщает о попытке вставить в дерево дублирующийся узел (при `cmp == 0`). Функция вставки нового элемента в список ничего подобного не делала, поскольку для этого следовало бы перебрать весь список, и операция вставки приобрела бы характер $O(n)$, а не $O(1)$. А вот в случае дерева, во-первых, такая проверка не требует дополнительных усилий, и, во-вторых, качество всей структуры данных пострадало бы от наличия дублирующихся записей. Правда, иногда бывает полезно разрешить вставку дублирующихся записей или же вообще игнорировать факт дублирования.

Функция `weprintf` представляет собой вариант `eprintf`. Она выводит сообщение об ошибке, в начале которого стоит слово `warning` (“предупреждение”), но в отличие от `eprintf` не завершает работу программы.

Дерево, в котором любой путь от корня к листу имеет примерно одинаковую длину, называется *сбалансированным*. Преимущество сбалансированного дерева заключается в том, что поиск по нему имеет характер $O(\log n)$, поскольку, как и в двоичном поиске, на каждом шаге отбрасывается половина оставшихся данных.

Если узлы добавляются в дерево по мере их поступления, дерево может оказаться несбалансированным; более того, оно может стать исключительно разбалансированным. Например, если элементы прибывают в отсортированном виде, то спуск всегда будет выполняться до самого низа одной из ветвей дерева, фактически представляя собой список по указателю `right`. Этот случай характеризуется всеми проблемами быстродействия, присущими спискам. Но если элементы поступают в случайном порядке, то подобная ситуация маловероятна и дерево будет более-менее сбалансированным.

Достаточно сложно реализовать такое дерево, которое гарантированно будет сбалансированным. Это одна из причин, по которой существует так много типов деревьев. Для целей нашего изложения мы оставим все эти вопросы в стороне и будем полагать, что поступающие данные достаточно стохастичны, чтобы поддерживать сбалансированность дерева.

Код функции поиска `lookup` похож на `insert`:

```
/* lookup: ищет имя name в дереве treep */
Nameval *lookup(Nameval *treep, char *name)
{
    int cmp;

    if (treep == NULL)
        return NULL;
    cmp = strcmp(name, treep->name);
    if (cmp == 0)
        return treep;
    else if (cmp < 0)
        return lookup(treep->left, name);
    else
        return lookup(treep->right, name);
}
```

Нужно отметить следующее по поводу функций `lookup` и `insert`. Во-первых, они очень напоминают реализации двоичного поиска, о которых упоминалось в начале этой главы. Это не случайно, поскольку в них заложена та же идея, что и в двоичном поиске ("разделяй и властвуй"), откуда происходит и быстродействие логарифмического порядка.

Во-вторых, эти функции являются рекурсивными. Если их переписать в итерационной форме, они станут еще более похожими на алгоритм двоичного поиска. Фактически итерационный вариант функции `lookup` можно получить одним изящным преобразованием из ее рекурсивной версии. Если элемент не найден, то последняя операция в `lookup` — это возвращение результата вызова себя самой, т.е. рекурсивный вызов выполняется в конце. Такой вызов легко преобразовать в итерационную форму, подставив нужные аргументы и передав управление в начало. Самый прямой способ — это воспользоваться оператором `goto`, но цикл `while` лучше с точки зрения стиля:

```
/* nrlookup: нерекурсивный поиск имени name в дереве treep */
Nameval *nrlookup(Nameval *treep, char *name)
{
    int cmp;

    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0)
            return treep;
        else if (cmp < 0)
            treep = treep->left;
        else
            treep = treep->right;
    }
```

```
        return NULL;
    }
```

Как только появляется возможность обхода дерева, дальнейшие операции получаются сами собой. Можно воспользоваться некоторыми методами работы со списками, например, написать общую функцию перебора узлов с вызовом другой заданной функции в каждом узле. Однако на этот раз встает вопрос выбора: когда выполнять операции над узлом и когда обрабатывать остальную часть дерева? Ответ зависит от характера данных, представляемых деревом. Если это упорядоченные данные наподобие дерева двоичного поиска, то левая половина всегда обходится раньше правой. Но иногда упорядоченность данных имеет нетривиальный характер, как, например, в фамильном дереве, и порядок обхода дерева зависит от представляемых взаимоотношений между узлами.

Симметричный (*in-order*) обход имеет место в том случае, когда операция выполняется после обхода левого поддерева и до обхода правого:

```
/* applyinorder: симметричное применение функции fn к treep */
void applyinorder(Nameval *treep,
        void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL)
        return;
    applyinorder(treep->left, fn, arg);
    (*fn)(treep, arg);
    applyinorder(treep->right, fn, arg);
}
```

Эта процедура применяется, если узлы необходимо обработать в порядке сортировки, например, для вывода их всех по порядку:

```
applyinorder(treep, printnv "%s: %x\n");
```

Она также обеспечивает рациональный метод сортировки: вставить все элементы в дерево, разместить в памяти массив подходящей длины, а затем скопировать узлы по порядку в массив симметричным обходом дерева.

При *концевом* (*post-order*) обходе дерева операция над узлом инициируется только после обхода всех его потомков:

```
/* applypostorder: концевой обход с вызовом fn */
void applypostorder(Nameval *treep,
        void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL)
        return;
    applypostorder(treep->left, fn, arg);
    applypostorder(treep->right, fn, arg);
    (*fn)(treep, arg);
}
```

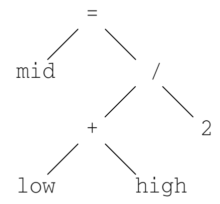
Концевой обход применяется в тех случаях, когда операция над узлом зависит от результата аналогичных операций над его потомками. Примеры таких операций — это вычисление высоты дерева (равной наибольшей из высот двух поддеревьев плюс единица), размещение графического изображения дерева на странице или экране в программе построения диаграмм (для этого нужно выделить каждому поддереву место в рабочем пространстве и суммировать их), измерение общей емкости дерева.

Третий способ обхода — *прямой* (*pre-order*) — применяется очень редко, так что не будем здесь тратить время на его обсуждение.

На практике деревья двоичного поиска применяются довольно редко, хотя B-деревья, характерные очень интенсивным ветвлением, используются для хранения информации на вспомогательных носителях. В повседневной работе программиста часто встречается такое применение деревьев, как разложение структуры оператора или выражения. Например, следующий оператор можно представить в виде *синтаксического дерева*:

```
mid = (low + high) / 2;
```

Синтаксическое дерево этого оператора показано на рисунке.



Чтобы проделать вычисления по даному дереву, следует выполнить его концевой обход, применяя соответствующие операции в каждом узле. Более подробно синтаксические деревья рассматриваются в главе 9.

Упражнение 2.11. Сравните быстродействие функций `lookup` и `nvlookup`. Какова разница между рекурсивной и итерационной формами?

Упражнение 2.12. Напишите функцию сортировки с симметричным обходом. Какой порядок по быстродействию имеет данная операция? При каких условиях она может работать плохо? Каковы ее характеристики по сравнению с алгоритмом быстрой сортировки и с библиотечными функциями?

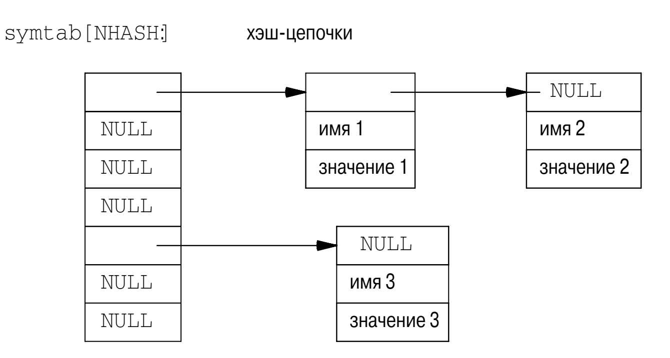
Упражнение 2.13. Придумайте и реализуйте набор тестов для проверки правильности функций работы с деревьями, рассмотренных выше.

2.9. Хэш-таблицы

Хэш-таблицы — это одно из величайших достижений в науке программирования. В них объединены черты массивов, списков и некоторых сложных математических методов. В результате получаются эффективные структуры для хранения и обработки динамических данных. Типичным применением хэш-таблиц являются *таблицы символов*, устанавливающие ассоциации между некоторыми значениями (*данными*) и элементами динамического набора строк (*ключей*). Почти в любом компиляторе

используется хэш-таблица, хранящая информацию о всех переменных компилируемой программы. Web-браузеры часто используют такие таблицы для учета просмотренных страниц, а программы связи с Internet прибегают к хэш-таблицам для кэширования доменных имен и IP-адресов.

Общая идея состоит в том, чтобы передать ключ в специальную *хэш-функцию* и получить *хэш-код*, значение которого принадлежало бы к ограниченному целочисленному диапазону, равномерно заполненному такими ключами. Затем полученный хэш-код используется для индексирования таблицы, в которой непосредственно хранится информация. В языке Java имеется стандартный интерфейс для работы с хэш-таблицами. В C и C++ имеется свой стиль их реализации. Обычно с каждым хэш-кодом, или ячейкой (*bucket*) таблицы, ассоциируется список элементов, соответствующих этому коду, как показано на рисунке.



На практике хэш-функция задается заранее, и для массива данных заблаговременно распределяется необходимый объем памяти (часто даже на этапе компиляции программы). Каждый элемент массива представляет собой список, который связывает элементы с общим хэш-кодом в цепочку. Другими словами, хэш-таблица из n элементов является массивом списков, средняя длина которых составляет n / (размер массива). Извлечение элемента из таблицы есть операция порядка $O(1)$ при условии, что хэш-функция подобрана удачно и списки не слишком разрастаются в длину.

Поскольку хэш-функция представляет собой массив списков, тип ее элементов должен быть тот же, что и у списка:

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next;      /* следующий в цепочке */
};

Nameval *symtab[NHASH]; /* таблица символов */
```

Для работы с отдельными цепочками ячеек применяются методы, рассмотренные в разделе 2.7. Если подобрана хорошая хэш-функция, то не будет никаких проблем: вычисляйте хэш-код, идите в нужную ячейку и перебирайте список, пока не найдете требуемый элемент. Ниже приведен код функции поиска и добавления элемента.

Если функция находит элемент в хэш-таблице, она возвращает указатель на него. Если элемент не найден и при этом установлен флаг создания `create`, то элемент добавляется в таблицу. Копия имени не создается; предполагается, что вызывающая функция сама позаботится об этом.

```
    /* lookup: находит имя name в таблице symtab, по необходимости
добавляя */
    Nameval* lookup(char *name, int create, int value)
    {
        int h;
        Nameval *sym;

        h = hash(name);
        for (sym = symtab[h]; sym != NULL; sym = sym->next)
            if (strcmp(name, sym->name) == 0)
                return sym;
        if (create) {
            sym = (Nameval *) emalloc(sizeof(Nameval));
            sym->name = name; /* размещается в другом месте */
            sym->value = value;
            sym->next = symtab[h];
            symtab[h] = sym;
        }
        return sym;
    }
```

Это сочетание поиска и создания нового элемента по мере необходимости широко распространено. Если не сочетать эти операции, работа будет дублироваться. Придется писать так:

```
    if (lookup("name") == NULL)
        additem(newitem("name", value));
```

В этом случае хэш-код вычисляется дважды.

Каким должен быть размер массива? Общая идея состоит в том, что массив должен быть достаточно велик, чтобы цепочка каждого хэш-кода содержала поменьше элементов, и операция поиска имела характер $O(1)$. Например, в компиляторе таблица может содержать несколько тысяч ячеек, поскольку большой файл исходного кода обычно содержит несколько тысяч строк, и ожидается, что новые идентификаторы будут встречаться с примерной частотой по одному на строку.

Теперь необходимо решить, что же будет вычислять хэш-функция `hash`. Функция должна работать по детерминистическому алгоритму, выдавать код достаточно быстро и при этом равномерно распределять данные по массиву. В одном из распространенных алгоритмов хэш-кодирования строк хэш-код строится путем прибавления каждого байта строки по очереди к уже накопленному хэш-коду, умноженному на некоторый коэффициент. Умножение позволяет распределить биты из нового байта по накопленному коду; в конце цикла кодирования получается хорошо перемешанная “каша” из байтов входных данных. Для строк в кодировке ASCII эмпирически выведенное значение коэффициента, используемого при таком хэш-кодировании, составляет от 31 до 37.

```
enum { MULTIPLIER = 31; }
/* hash: вычисляет хэш-код строки */
unsigned int hash(char *str)
{
    unsigned int h;
    unsigned char *p;

    h = 0;
    for (p = (unsigned char *) str; *p != '\0'; p++)
        h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

В этой операции явным образом используются символы типа `unsigned char` (без знака), поскольку стандарт C или C++ не гарантирует наличие или отсутствие знака у символьных переменных, а хэш-код должен быть положительным.

Хэш-функция возвращает остаток от деления накопленного результата на длину массива. Если хэш-функция распределяет ключи равномерно, точный размер массива не имеет значения. Однако стопроцентную надежность хэш-функции гарантировать трудно, и даже наилучшие функции могут столкнуться с проблемами при обработке некоторых наборов входных данных. Поэтому рекомендуется сделать размер массива равным простому числу и тем самым дополнительно подстраховаться, поскольку в этом случае размер массива, коэффициент хэш-кодирования и вероятные значения данных не будут иметь общего делителя.

Эксперименты показывают, что для очень широкого круга строк трудно построить хэш-функцию, которая бы работала заметно эффективнее, чем приведенная выше. Зато очень легко сконструировать такую, которая бы работала хуже. В ранней версии Java существовала хэш-функция для кодирования строк, которая работала тем эффективнее, чем длиннее была строка. Эта хэш-функция экономила время, принимая в расчет только 8 или 9 символов через равномерные интервалы, начиная с начала, если кодируемая строка была длиннее 16 символов. К сожалению, хоть эта функция и работала быстро, ее плохие статистические показатели сводили на нет все преимущества быстродействия. Пропуская фрагменты строки, она игнорировала наиболее значимые ее части. Например, имена файлов начинаются с длинных, практически одинаковых префиксов — имен каталогов — и отличаются только последними несколькими символами (`.java` или `.class`). URL-адреса обычно начинаются с `http://www.`, а заканчиваются `.html`, имея тенденцию отличаться только в середине. Таким образом, хэш-функция часто принимала во внимание только неизменную часть строки и в результате строила длинные списки элементов в ячейках, снижавшие скорость поиска. Проблема была решена заменой этого алгоритма на другой, аналогичный показанному выше (с коэффициентом 37), который принимал во внимание все символы строки.

Хэш-функция, хорошо работающая с данными одного вида (например, короткими именами переменных), может оказаться неудачной в работе с другими (такими как URL-адреса), поэтому хэш-функцию для своей программы следует тестировать на типичных наборах входных данных. Хорошо ли она кодирует короткие строки? А длинные? А строки одинаковой длины с небольшими отличиями?

Строки — это не единственные объекты, которые можно организовывать в хэш-таблицы (хэш-кодировать). Например, в программе физического моделирования и расчета можно хэш-кодировать трехмерные координаты частиц, тем самым организуя хранилище данных в виде линейной таблицы порядка $O(\textit{количество частиц})$ вместо трехмерного массива порядка $O(\textit{размер}X \times \textit{размер}Y \times \textit{размер}Z)$.

Одно из замечательных применений хэш-кодирования можно найти в программе Джерарда Хольцмана (Gerard Holzmann) под названием Supertrace, предназначенной для анализа протоколов и систем параллельной обработки данных. Программа Supertrace получает на вход полную информацию о всех возможных состояниях анализируемой системы и хэширует ее, чтобы получить адрес единственного бита памяти. Если этот бит установлен, значит, состояние уже было зарегистрировано ранее; если сброшен, то нет. В программе Supertrace используется хэш-таблица длиной во много мегабайт, но в каждой ячейке хранится всего один бит. Цепочки (списки) данных не создаются; если вдруг два состояния *конфликтуют*, получая один и тот же хэш-код, программа просто игнорирует этот факт. Программа полагается на низкую вероятность подобного конфликта (эта вероятность не обязана быть нулевой, поскольку алгоритм работы Supertrace носит вероятностный, а не детерминистический характер). Поэтому хэш-функция в ней написана особенно тщательно, с использованием *циклического избыточного контроля* (*cyclic redundancy check*), который позволяет получить очень равномерную “смесь” исходных данных.

Хэш-таблицы прекрасно подходят для организации таблиц символов, поскольку обеспечивают доступ к любому элементу за время порядка $O(1)$. Но у них есть и ряд недостатков. Если хэш-функция написана плохо или размер таблицы недостаточен, то списки в ячейках вырастают до нерациональной длины. Поскольку эти списки не отсортированы, время обращения к данным возрастает до $O(n)$. Элементы нельзя непосредственно расположить в отсортированном порядке, но можно подсчитать их, выделить массив нужной длины, заполнить его указателями на элементы и отсортировать их. И все же при правильной организации хэш-таблиц им нет равных среди других структур данных по таким показателям, как объем операций, требуемый для поиска, вставки и удаления элементов.

Упражнение 2.14. Наша хэш-функция имеет довольно общий характер и удобна при работе со строками. Однако с некоторыми исходными данными она может справляться недостаточно эффективно. Сконструируйте набор данных, который бы заставил эту функцию работать плохо. Насколько трудно построить такой набор для различных значений `NHASH`?

Упражнение 2.15. Напишите функцию для обращения к последовательным элементам хэш-таблицы в несортированном порядке.

Упражнение 2.16. Модифицируйте функцию `lookup` так, чтобы при превышении средней длиной списка некоторого порога `x` массив расширялся бы автоматически с коэффициентом пропорциональности `y` и чтобы хэш-таблица подвергалась перестройке.

Упражнение 2.17. Разработайте хэш-функцию для хранения координат точек в двумерном пространстве. Насколько легко адаптировать вашу функцию к изменениям типа координат (например, от целочисленных к вещественным), системы координат (от декартовой к полярной) или размерности (от двух к более высокой)?

2.10. Резюме

Алгоритм для решения своей задачи следует выбирать в несколько этапов. Вначале проанализируйте потенциально пригодные алгоритмы и структуры данных. Оцените, какой объем данных придется обрабатывать программе. Если в задаче участвует сравнительно немного данных, ограничьтесь простыми методами; если объем данных может возрасти в будущем, отбросьте методы, не приспосабливающиеся к увеличению размерности задачи. Затем выберите средство языка или стандартной библиотеки для решения поставленной проблемы. Если такового не существует, постарайтесь написать (или заимствовать) короткую, простую и легко понятную программную реализацию решения. Протестируйте ее. Только если тесты покажут, что она работает слишком медленно, следует переходить к более усовершенствованным методам.

Хотя существует множество структур данных, и многие из них играют критическую роль для быстродействия программ, на практике большинство программных систем основано на использовании массивов, списков, деревьев и хэш-таблиц. Каждая из этих структур поддерживает определенный набор базовых операций, таких как создание нового элемента; поиск, добавление, удаление элемента; применение некоторых операций ко всем элементам сразу.

Каждая из операций имеет свое характерное время выполнения, которое часто определяет, насколько хорошо конкретный тип данных (или его реализация) подходит для того или иного приложения. Например, массивы предоставляют такой доступ к своим элементам, время которого не зависит от их количества, но зато неспособны достаточно гибко расти или уменьшаться в размерах. В списках удобно выполнять операции добавления и удаления, но для обращения к конкретному элементу требуется порядка $O(n)$ операций. Деревья и хэш-таблицы являются разумным компромиссом: быстрый доступ к отдельным элементам сочетается с простыми механизмами расширения, по крайней мере при соблюдении некоторого критерия сбалансированности.

Для специализированных задач часто создаются более сложные и изощренные структуры данных, но перечисленного набора вполне достаточно для программной реализации подавляющего большинства прикладных задач.

Дополнительная литература

Доступное изложение большого количества разнообразных алгоритмов можно найти в серии книг: Bob Sedgewick, *Algorithms* (Addison-Wesley). Третье издание книги *Algorithms in C++* из этой серии (1998 г.) содержит полезный материал по хэш-функциям и вопросам объема хэш-таблиц. Надежным источником строго и последовательно проанализированных алгоритмов является книга Donald Knuth, *The Art of Computer Programming* (Addison-Wesley). В 3-м томе этой книги (2-е издание, 1998 г.) как раз рассматриваются алгоритмы сортировки и поиска[1].

Программа Supertrace описана в книге Gerard Holzmann, *Design and Validation of Computer Protocols* (Prentice Hall, 1991).

Построение эффективного и устойчивого алгоритма быстрой сортировки рассматривается в статье J. Bentley, D. McIlroy, “Engineering a sort function”, *Software — Practice and Experience*, 1993, **23**, 1, p. 1249–1265.

---

[1] Все три книги Дональда Кнута “Искусство программирования” вышли на русском языке в ИД “Вильямс” в 2000 году. — *Прим. ред.*

3

Проектирование и реализация

Покажите мне ваши блок-схемы, спрятав таблицы данных, — и я по-прежнему буду теряться в догадках. Покажите таблицы — и блок-схемы, как правило, не понадобятся, поскольку будут очевидны.

Фредерик П. Брукс-мл. "Мифический человеко-месяц"
(Frederick P. Brooks, Jr., The Mythical Man Month*)*

Перефразируя эпиграф, в котором приводится цитата из классической книги Ф. Брукса, можно утверждать, что проектирование структур данных является решающим элементом при разработке программ. Как только структуры данных окончательно определены, в алгоритме все обычно становится на свои места и программная реализация уже не представляет никакой сложности.

Конечно, эта точка зрения несколько упрощена, но все же недалека от действительности. В предыдущей главе рассматривались основные структуры данных, являющиеся строительным материалом для большинства программ. В этой главе мы скомбинируем несколько подобных структур в ходе проектирования и реализации небольшой программы. Будет показано, как суть задачи влияет на применяемые структуры данных и как последующее написание кода облегчается тщательным выбором объектов для хранения данных.

Один из аспектов этой точки зрения состоит в том, что выбор языка программирования не так уж существен при проектировании программы в целом. Вначале мы спроектируем абстрактную реализацию алгоритма, а затем напишем ее на языках C, Java, C++, Awk и Perl. При сравнении этих реализаций станет ясно, в чем особенности конкретных языков помогают и в чем мешают воплощению алгоритма в программу, а также в каких аспектах они несущественны. Выбор языка программирования придает программе тот или иной колорит, но редко влияет на ее принципиальную структуру.

Выбранная для этой цели задача довольно необычна, но самая общая ее форма такая же, как и у других программ: на вход поступают одни данные, на выходе получаются другие, и для превращения одних в другие необходимо применить набор нетривиальных операций.

Мы хотим сгенерировать случайный текст на английском языке, который можно было бы прочитать. Если генерировать слова или буквы по счетчику случайных чисел, в результате получается бессмыслица. Например, если программа выдает наугад

выбранные буквы (а также пробел для отделения одного слова от другого), то может получиться примерно следующее:

```
xptmxgn xusaja  afqnzqxl  lhidlwcd rjdjuvpydrlwnjy
```

Это мало напоминает текст. Если ассоциировать с буквами весовые коэффициенты по частоте их употребления в английском языке, то получится нечто вроде этого:

```
idtefoae tcs trder jcii ofdslnqetacp t ola
```

Нельзя сказать, что этот результат чем-то лучше предыдущего. Также не слишком осмысленным получается текст, если составить его из слов, наугад выбранных из словаря:

```
polydactyl equatorial splashily jowl verahdah circumscribe
```

Для получения более качественных результатов нам необходима более структурированная статистическая модель, например, распределение частот словосочетаний. Но где взять такую статистику? Можно, конечно, взять большой объем английских текстов и подробно изучить их, но есть более легкий подход, сочетающий приятное с полезным. Ключевое наблюдение состоит в том, что можно взять любой текст и построить статистическую модель *именно этого текста*, а затем генерировать на его основе случайный текст со статистическими свойствами, подобными оригиналу.

3.1. Цепь Маркова

Поставленная задача обработки данных изящно решается с помощью алгоритма, в котором используется *цепь Маркова*. Представим себе входные данные как цепочку перекрывающихся словосочетаний. Данный алгоритм делит каждую фразу на две части: *префикс* из нескольких слов и суффикс из одного слова, следующий за *префиксом*. Рассматриваемый алгоритм генерирует фразы на выходе, случайным образом выбирая для определенного префикса следующий за ним суффикс. Выбор делается на основе статистики — в данном случае статистики исходного текста. Для этой цели вполне подходят сочетания из трех слов, т.е. по префиксу из двух слов выбирается третье слово-суффикс:

```
поместить в w1 и w2 первые два слова текста;
вывести w1 и w2;
цикл:
    случайно выбрать w3 из набора суффиксов
    к префиксу w1 и w2;
    заменить w1 и w2 на w2 и w3;
    повторить цикл;
```

Для иллюстрации этого подхода предположим, что необходимо сгенерировать случайный текст на основе нескольких адаптированных фраз из оригинального текста эпиграфа к данной главе, используя префиксы из двух слов:

```
Show your flowcharts and conceal your tables and I will be mystified.
Show your tables and your flowcharts will be obvious. (end)
```

Вот некоторые пары слов из исходных данных и следующие за ними слова.

Префиксы из двух слов	*Слова-суффиксы*
Show your	flowcharts tables
your flowcharts	and will
flowcharts and	conceal
flowcharts will	be
your tables	and and
will be	mystified. obvious.
be mystified	Show
be obvious.	(*end*)

Марковский алгоритм начинает свою работу по обработке этого текста с вывода слов `Show your`, а затем случайным образом выбирает либо `flowcharts`, либо `tables`. Если выбирается первый вариант, то префикс приобретает вид `your flowcharts` и следующее слово должно быть `and` или `will`. Если выбирается вариант `tables`, то следующим словом будет `and`. Так продолжается до тех пор, пока не будет сгенерировано достаточно текста или пока в качестве суффикса не встретится маркер (*end*).

Наша программа будет считывать фрагмент английского текста и использовать марковский алгоритм для генерирования нового текста на основе частот появления словосочетаний фиксированной длины. Количество слов в префиксе, а в нашем примере их два, будет являться параметром задачи. Чем короче префикс, тем более бессвязным оказывается новый текст; чем этот префикс длиннее, тем больше программа склонна повторять текст практически дословно. Для англоязычного текста выбор третьего слова по двум предыдущим является хорошим компромиссом. Это позволяет сохранить колорит исходного текста, добавив к нему новые причудливые штрихи.

Что такое слово? Очевидный ответ таков: это последовательность букв алфавита. Однако желательно оставить в тексте его пунктуацию, поэтому следует считать слова “`words`” и “`words.`” различными. Таким образом улучшается качество сгенерированного текста, поскольку в нем сохраняются знаки препинания, а значит, и до некоторой степени грамматика. Это обстоятельство также влияет на выбор слов, хотя часто приводит к незакрытым кавычкам или скобкам и т.п. Поэтому определим “слово” как строку символов, выделенную пустым пространством; это определение не накладывает никаких ограничений на входной язык и оставляет на месте знаки препинания, стоящие после слов. Поскольку в языках программирования обычно имеются средства для разделения текста на такие “слова”, этот подход можно легко реализовать.

В силу выбранного метода все отдельные слова, а также словосочетания из двух и трех слов, возникающие на выходе программы, уже встречались во входных данных. Однако наберется немало заново синтезированных фраз из четырех и более слов. Ниже приведено несколько предложений, выданных той программой, которую мы еще только собираемся написать в этой главе. Исходным материалом послужила глава 7 из книги Э. Хэмингуэя “И восходит солнце”.

> As I started up the undershirt onto his chest black, and big stomach muscles bulging under the light. "You see them?" Below the line where his ribs stopped were two raised white welts. "See on the forehead." "Oh, Brett, I love you." "Let's not talk. Talking's all bilge. I'm going away tomorrow." "Tomorrow?" "Yes. Didn't I say so? I am." "Let's have a drink, then." (Как только я рубашку на его грудь черную, и могучие брюшные мышцы вздувавшиеся в свете. "Видите?" Пониже того места, где кончались ребра, было два выпуклых белых шрама. "Смотри в лоб." "Ах, Брет, я так тебя люблю." "Долой разговоры. Разговоры - чушь. Я завтра уезжаю." "Завтра?" "Да. Разве я не говорила? Уезжаю." "Тогда пойдем выпьем.")

Здесь нам повезло — знаки препинания сохранились. Но так происходит не всегда.

3.2. Выбор структур данных

С каким объемом входных данных придется иметь дело? Каким быстродействием должна отличаться программа? Кажется разумным, что она должна уметь проанализировать целую книгу, поэтому следует быть готовыми к чтению около $n = 100\,000$ слов. На выходе будет сгенерировано несколько сотен или тысяч слов, и программе для этого должно понадобиться несколько секунд, а не минут. Имея 100 000 слов исходного текста, не следует выбирать слишком примитивный алгоритм, иначе программа не будет работать достаточно быстро.

Используемый марковский алгоритм должен проанализировать все входные данные, прежде чем он начнет генерировать что-либо на выходе. Поэтому придется хранить сразу весь объем исходных данных в какой-то подходящей для этого форме. Один из простейших вариантов — это поместить весь исходный текст в длинную строку. Но нам, очевидно, необходимо иметь дело с текстом, разбитым на слова. Если сохранить его в виде массива указателей на слова-строки, то генерировать выходные данные будет достаточно просто: для вывода одного слова перебираем входной текст в поисках суффиксов, следующих за только что сгенерированным префиксом, а затем выбираем один из их числа наугад. Однако это означает, что придется перебирать все 100 000 исходных слов для генерирования каждого слова на выходе. Если нужно вывести 1000 слов, это будет означать сотни миллионов операций сравнения строк, и о быстродействии придется забыть.

Следующий вариант — это хранить только неповторяющиеся слова из входных данных вместе со списками мест, в которых они встречаются в тексте, чтобы поиск следующих за ними слов выполнялся быстрее. Для этого можно было бы воспользоваться такой таблицей, какая описана в главе 2, но она не вполне подходит для нужд нашего алгоритма — быстрого поиска всех суффиксов по заданному префиксу.

Нам нужна структура данных, которая бы наилучшим образом представляла префикс и ассоциированные с ним суффиксы. Программа будет работать в два прохода: на первом проходе будет строиться структура данных, представляющая словосочетания, а на втором — генерироваться случайный выходной текст. На обоих проходах необходимо уметь быстро разыскивать любой конкретный префикс: сначала для обновления списка его суффиксов, затем для случайного выбора из этого списка.

Напрашивается хэш-таблица, в которой префиксы играли бы роль ключей, а значения состояли бы из наборов соответствующих им суффиксов.

Для определенности выберем префикс из двух слов, т.е. каждое слово на выходе программы будет определяться парой предшествующих ему слов. Количество слов в префиксе не влияет на структуру программы, и программа должна быть способна обрабатывать префиксы любой длины; выбор длины просто делает наше изложение более конкретным. Назовем совокупность префикса и соответствующего ему набора возможных суффиксов *состоянием*; это стандартный термин в описаниях алгоритмов, основанных на цепях Маркова.

Имея префикс, необходимо сохранить все его суффиксы, чтобы обращаться к ним впоследствии. Суффиксы никак не упорядочены и добавляются по одному за раз. Заранее неизвестно, сколько всего их будет в каждом случае, поэтому нужна эффективно и быстро расширяемая структура данных, например список или динамический массив. При генерировании текста необходимо наугад выбирать один суффикс из их набора, ассоциированного с данным префиксом. Элементы структуры исходных данных никогда не удаляются.

Как быть, если словосочетание встречается несколько раз? Например, словосочетание "`встретиться дважды`" может встретиться дважды, а словосочетание "`только однажды`" — только однажды. Эту задачу можно решить, либо помещая суффикс "`дважды`" дважды в список суффиксов, либо помещая его только один раз, но придав ему счетчик со значением 2. Мы попробовали и так и этак. Без счетчика оказалось легче, поскольку при добавлении суффикса не требуется проверять, есть ли он уже в списке, а разница в быстродействии была несущественной.

Подведем итоги. Каждое состояние включает в себя префикс и список суффиксов. Эта информация помещается в хэш-таблицу, ключами которой служат префиксы. Каждый префикс представляет собой набор слов фиксированной длины. Если суффикс встречается в связи с конкретным префиксом несколько раз, каждое его вхождение включается в список отдельно.

Дальше следует принять решение, как представлять сами слова. Самый легкий способ — это хранить их в виде отдельных строк. Поскольку в большинстве текстов одни и те же слова встречаются множество раз, наверное, стоило бы сэкономить место в памяти, заведя вторую хэш-таблицу отдельных слов, чтобы хранить каждое из них в единственном экземпляре. Это, вероятно, ускорило бы хэш-кодирование префиксов, поскольку можно было бы сравнивать указатели, а не отдельные символы — уникальные строки будут иметь уникальные адреса. Реализацию этого подхода мы оставляем читателю в качестве упражнения; пока же все слова будут храниться в отдельных строках.

3.3. Построение структуры данных на С

Начнем с реализации программы на языке С. Первым делом нужно объявить несколько констант.

```
enum {
    NPREF  = 2,     /* количество слов в префиксе */
    NHASH  = 4093,  /* размер массива состояний */
    MAXGEN = 10000  /* максимальный объем текста на выходе */
```

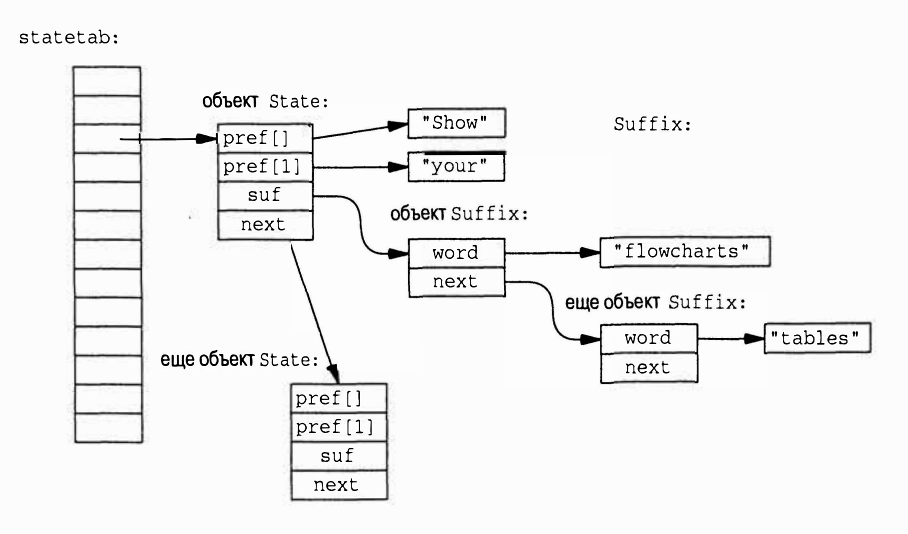
В этой декларации объявляются количество слов (NPREF) в префиксах, размер массива хэш-таблицы (NHASH) и верхний предел для количества слов в генерируемом тексте (MAXGEN). Если NPREF представляет собой константу, определяемую до компиляции, а не переменную, то управление данными значительно упрощается. Размер массива выбран достаточно большим, поскольку на вход программы предполагается подавать большие документы, такие как целые книги. Выбрано значение NHASH = 4093, поскольку если входные данные содержат 10 000 различных префиксов (пар слов), то средняя длина цепочки в хэш-таблице совсем невелика — два-три префикса. Чем больше размер, тем короче средняя длина такой цепочки и тем быстрее осуществляется поиск в таблице. Наша программа не имеет особого практического значения (это просто игрушка), поэтому ее быстродействие не играет роли, но если сделать массив слишком коротким, то программа вообще не справится с входными данными за сколько-нибудь разумное время. С другой стороны, если слишком раздуть размеры массива, он может не поместиться в имеющуюся память.

Префикс можно хранить в виде массива слов. Элементы хэш-таблицы будут иметь структурный тип State, в котором с каждым префиксом ассоциируется список суффиксов (элементов типа Suffix):

```
typedef struct State State;
typedef struct Suffix Suffix;
struct State {  /* префикс + список суффиксов */
    char    *pref[NPREF];   /* слова-префиксы */
    Suffix  *suf;           /* список суффиксов */
    State   *next;          /* следующий элемент в таблице */
};
struct Suffix { /* список суффиксов */
    char    *word;          /* суффикс */
    Suffix  *next;          /* следующий элемент в списке */
}

State   *statetab[NHASH];   /* хэш-таблица состояний */
```

Символически можно изобразить эту структуру данных следующим образом:



Необходимо построить хэш-функцию для префиксов, представляющих собой массивы строк. Нетрудно модифицировать строковую хэш-функцию из главы 2 для перебора всех строк в массиве, таким образом выполняя хэш-кодирование конкатенации (сцепления) строк:

```
/* hash: вычисляет хэш-код для массива из NPREF строк */
unsigned int hash(char *s[NPREF]);
{
    unsigned int h;
    unsigned char *p;
    int i;

    h = 0;
    for (i = 0; i < NPREF; i++)
        for (p = (unsigned char *) s[i]; *p != '\0'; p++)
            h = MULTIPLIER * h + *p;
    return h % NHASH;
}
```

Выполнив аналогичную модификацию функции поиска по таблице, получаем окончательную реализацию требуемой хэш-таблицы:

```
/* lookup: ищет префикс или создает его по запросу. */
/*  возвращает указатель на префикс; NULL в случае неудачи. */
/*  при создании не вызывается strdup, поэтому строки нельзя
     изменять */
State* lookup(char *prefix[NPREF], int create)
{
    int i, h;
    State sp;

    h = hash(prefix);
    for (sp = statetab[h]; sp != NULL; sp = sp->next) {
        for (i = 0; i < NPREF; i++)
            if (strcmp(prefix[i], sp->pref[i]) != 0)
                break;
        if (i == NPREF)     /* префикс найден */
            return sp;
    {
    if (create) {
        sp = (State *) emalloc(sizeof(State));
        for (i = 0; i < NPREF; i++)
            sp->pref[i] = prefix[i];
        sp->suf = NULL;
        sp->next = statetab[h];
        statetab[h] = sp;
    }
    return sp;
}
```

Заметьте, что функция `lookup` не делает копию поступающей строки при создании нового состояния; она всего лишь помещает указатель на нее в `sp->pref[]`. Функции, из которых вызывается `lookup`, должны сами гарантировать, что строковая информация не будет впоследствии затерта. Например, если строка находится в буфере ввода-вывода, то перед вызовом `lookup` ее необходимо скопировать в ста-

ционарное место в памяти, чтобы при последующих операциях ввода не были затерты данные, на которые уже имеется ссылка в хэш-таблице. В программах часто возникает необходимость принять решение, какой из модулей будет владеть и распоряжаться тем или иным ресурсом общего пользования. Эта тема будет рассмотрена более подробно в следующей главе.

Далее необходимо построить хэш-таблицу по мере считывания файла исходных данных:

```
/* build: считывает входные данные, строит таблицу префиксов */
void build(char *prefix[NPREF], FILE *f)
{
    char buf[100], fmt[10];

    /* строится строка формата; %s затерла бы буфер */
    sprintf(fmt, "%%%ds", sizeof(buf)-1);
    while (fscanf(f, fmt, buf) != EOF)
        add(prefix, estrdup(buf));
}
```

Интересный вызов функции `sprintf` призван обойти известную проблему с функцией `fscanf`, которая в других случаях идеально подходит для такой работы. Вызов `fscanf` со спецификацией формата `%s` приводит к считыванию следующего слова (строки, окруженной пробелами) из файла в буфер, но при этом никак не ограничивается длина слова. Слишком длинное слово может переполнить буфер ввода, и последствия будут непредсказуемыми. Если объявить буфер длиной 100 символов (а это намного больше, чем длина слова в любом английском тексте), то можно воспользоваться спецификацией `%99s`, оставив один байт для завершающего нуля (`'\0'`). В этом случае функция `fscanf` прекращает ввод не позже чем через 99 символов. Длинное слово может оказаться разбитым на части, что неудобно, однако совершенно безопасно. Можно было бы объявить так:

```
?   enum { BUFSIZE = 100 };
?   char    fmt[] = "%99s"; /* BUFSIZE - 1 */
```

Но в этом случае одной ключевой величине — размеру буфера — соответствуют сразу две объявленные переменные, и при этом возникает необходимость следить за их соответствием друг другу. Задача решается раз и навсегда введением динамической строки формата, которая строится по ходу функции с помощью вызова `sprintf`. Именно этот подход мы и применили.

Функция `build` принимает два аргумента: массив `prefix`, содержащий предыдущие `NPREF` слов из входного файла, и указатель типа `FILE`. В свою очередь, она передает в функцию `add` массив `prefix` и копию поступившего на вход слова. Функция `add` добавляет новый элемент в хэш-таблицу и передвигает префикс на одну позицию вперед:

```
/* add: пополняет список суффиксов, обновляет префикс */
void add(char *prefix[NPREF], char *suffix)
{
    State *sp;

    sp = lookup(prefix, 1);  /* создаем, если не нашли */
    addsuffix(sp, suffix);
```

```
        /* сдвиг слов в префиксе */
        memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
        prefix[NPREF-1] = suffix;
    }
```

Вызов функции memmove — это стандартная конструкция для удаления элементов из массива. При этом элементы с номерами от 1 до NPREF-1 в префиксе сдвигаются в позиции от 0 до NPREF-2 соответственно, тем самым удаляя первое слово префикса и освобождая последнюю позицию для нового слова.

Функция addsuffix добавляет новый суффикс в список:

```
    /* addsuffix: добавляет суффикс к состоянию; суффикс изменять
нельзя */
    void addsuffix(State *sp, char *suffix)
    {
        Suffix *suf;

        suf = (Suffix *) emalloc(sizeof(Suffix));
        suf->word = suffix;
        suf->next = sp->suf;
        sp->suf = suf;
    }
```

Итак, операция обновления состояния разнесена по двум функциям: функция add выполняет подготовительную работу по добавлению суффикса к префиксу, тогда как функция addsuffix выполняет специфическую проблемно-ориентированную операцию по добавлению слова к списку суффиксов. Функция add используется в функции build, а вот addsuffix вызывается только из add. Таким образом, addsuffix относится к числу низкоуровневых подробностей реализации, которые могут со временем измениться. Именно поэтому лучше иметь для этой цели отдельную функцию, пусть даже она и вызывается всего в одном месте.

3.4. Генерирование выходных данных

Разработав структуру данных, принимаемся за генерирование текста. Основная идея не изменилась: имея префикс, наугад выбираем один из суффиксов, выводим его и передвигаем префикс по тексту. Это стационарное состояние алгоритма, однако необходимо еще решить, с чего начинать и чем заканчивать его работу. Начать работу легко, если вначале запомнить слова самого первого префикса. Закончить тоже нетрудно — для этого нужно всего лишь иметь завершающее слово, маркер конца текста. После всех слов исходного текста можно поставить “слово”, которое гарантированно не может появиться ни в одном тексте:

```
    build(prefix, stdin);
    add(prefix, NONWORD);
```

Строка NONWORD должна содержать нечто такое, чего не бывает в обычном тексте на английском языке. Поскольку слова отделяются друг от друга во входном потоке символами свободного пространства, один из таких символов подойдет в качестве маркера конца. Это может быть символ конца строки:

```
    char NONWORD[] = "\n";  /* не бывает словом */
```

Еще одна потенциальная проблема: а что если текста на входе недостаточно, чтобы начать работу алгоритма? Имеются два подхода к проблемам подобного рода: либо завершить работу при недостатке исходных данных, либо сделать так, чтобы их заведомо было достаточно, и не заниматься проверкой этого факта. Второй подход вполне применим к нашей задаче.

Построение таблицы и генерирование текста можно начать с искусственно построенного префикса, тем самым гарантируя наличие данных для старта алгоритма. Для нулевой итерации цикла заполним префикс строками NONWORD и получим то дополнительное преимущество, что первое слово входного файла будет уже являться *суффиксом* фиктивного префикса, так что цикл генерации текста должен будет выводить только суффиксы без единого префикса.

Чтобы не получить слишком длинного текста на выходе, можно завершить работу алгоритма либо по достижении заданного количества слов, либо при появлении маркера NONWORD во входном потоке. Которое из событий произойдет раньше, то и будет критерием окончания алгоритма.

Добавление нескольких строк NONWORD в конец исходных данных существенно упрощает основные рабочие циклы программы. Это пример техники программирования, состоящей в добавлении *символов-разделителей* к исходным данным для обозначения их границ.

По возможности следует стараться обработать все нестандартные или исключительные ситуации, а также особые входные данные. Код должен быть написан как можно проще и строже, чтобы корректно реагировать на любые возможные эксцессы.

Функция generate реализует тот алгоритм, набросок которого был дан вначале. В нем генерируется одно слово на каждую выходную строку; затем с помощью текстового редактора можно собрать эти слова в более длинные строки. В главе 9 демонстрируется несложная процедура форматирования под названием fmt, вполне пригодная для этой цели.

Если в начале и в конце данных используются маркеры NONWORD, функция generate запускается и завершается вполне корректно:

```
/* generate: генерирует текст по одному слову в строке */
void generate(int nwords)
{
    State *sp;
    Suffix *suf;
    char *prefix[NPREF], *w;
    int i, nmatch;

    for (i = 0; i < NPREF; i++) /* переустановка префиксов */
        prefix[i] = NONWORD;

    for (i = 0; i < nwords; i++) {
        sp = lookup(prefix, 0);
        nmatch = 0;
        for (suf = sp->suf; suf != NULL; suf = suf->next)
            if (rand() % ++nmatch == 0) /* prob = 1/nmatch */
                w = suf->word;
        if (strcmp(w, NONWORD) == 0)
            break;
        printf("%s\n", w);
```

```
            memmove(prefix, prefix+1, (NPREF-1)*sizeof(prefix[0]));
            prefix[NPREF-1] = w;
        }
    }
```

Обратите внимание на алгоритм выбора одного из элементов наугад в условиях, когда неизвестно их общее количество. В переменной nmatch подсчитывается количество совпадений по мере прохождения списка. В следующем выражении переменная nmatch инкрементируется, а само выражение является истинным с вероятностью 1/nmatch:

```
rand() % ++nmatch == 0
```

Таким образом, первый совпадающий элемент выбирается с вероятностью 1, второй заменит его с вероятностью 1/2, третий встанет на его место с вероятностью 1/3 и т.д. На любом шаге алгоритма каждый из k уже перебранных элементов был выбран с вероятностью $1/k$.

Вначале в массив prefix заносятся фиксированные начальные значения, что гарантирует инициализацию хэш-таблицы. Первые найденные значения типа Suffix будут представлять собой первые слова текстового документа, поскольку только они следуют за начальным пустым префиксом. После этого суффиксы станут появляться на выходе в случайном порядке. В цикле вызывается функция lookup, разыскивающая в хэш-таблице соответствующие суффиксы для текущего префикса, затем из этих суффиксов случайным образом выбирается один, он выводится в выходной поток, и префикс сдвигается на слово вперед по тексту.

Если выбранный суффикс представляет собой строку NONWORD, то работа окончена, поскольку мы пришли к состоянию конца входного потока. Если суффикс не является строкой NONWORD, то выводим его, отбрасываем первое слово префикса вызовом функции memmove, назначаем суффикс последним словом префикса и переходим к началу цикла.

Теперь соберем все наши наработки в единое целое с помощью функции main, считывающей данные из стандартного потока ввода и генерирующей не более заданного количества слов:

```
    /* markov main: генератор случайного текста по марковскому
алгоритму */
    int main(void)
    {
        int i, nwords = MAXGEN;
        char *prefix[NPERF];        /* текущие префиксы */

        for (i = 0; i < NPERF; i++) /* инициализация префиксов */
            prefix[i] = NONWORD;
        build(prefix, stdin);
        add(prefix, NONWORD);
        generate(nwords);
        return 0;
    }
```

На этом наша C-реализация алгоритма закончена. В конце этой главы мы еще вернемся к ней, сравнивая работу программ на разных языках. Величайшие преимущества C состоят в том, что программист имеет полный контроль над реализаци-

ей алгоритма и получающиеся программы, как правило, работают очень быстро. Но платить за это приходится изрядным объемом дополнительной работы по распределению и освобождению памяти, явному созданию хэш-таблиц и связанных списков и т.п. Язык C — это остро отточенный инструмент, которым можно изваять элегантное произведение искусства, но можно и накромсать бесформенное месиво.

Упражнение 3.1. Алгоритм для выбора случайного элемента из списка неизвестной длины полагается на функцию-генератор случайных чисел, от качества работы которого зависит весь алгоритм. Продумайте и проведите эксперимент по анализу того, насколько хорошо этот метод работает на практике.

Упражнение 3.2. Если каждое слово из входного потока помещать во вторую хэш-таблицу, то весь текст будет храниться без дублирования и таким образом будет сэкономлено место в памяти. Проведите эксперименты с несколькими документами, чтобы оценить, насколько велика эта экономия. При такой организации данных в хэш-цепочках префиксов можно сравнивать указатели, а не строки, и быстродействие должно увеличиться. Реализуйте эту версию программы и оцените разницу как в быстродействии, так и в потреблении памяти.

Упражнение 3.3. Удалите операторы, помещающие разделительные строки `NONWORD` в начале и в конце входных данных, а затем модифицируйте функцию `generate` так, чтобы она стартовала и завершалась корректно без них. Убедитесь, что программа правильно генерирует текст на основе исходных данных из 0, 1, 2, 3 и 4 слов. Сравните эту версию с той, в которой используются строки-разделители.

3.5. Java

Для второй реализации марковского алгоритма будет использоваться язык Java. Объектно-ориентированные языки наподобие Java стимулируют программиста тщательно продумывать способы взаимодействия (интерфейсы) компонентов программы. Такие компоненты инкапсулируются в самостоятельные группировки данных, известные под названием объектов или классов, а с ними ассоциируются функции, именуемые методами.

В языке Java имеется более богатая библиотека, чем в C, в том числе набор *классов-контейнеров* для группировки элементарных объектов разными способами. Один из примеров — это класс `Vector`, который реализует динамически растущий массив и способен хранить объекты любого производного от `Object` типа. Есть также класс `Hashtable`, в котором можно хранить объекты одного типа, извлекая их по ключам — объектам другого типа.

В нашей задаче префиксы и суффиксы вполне естественно хранить в объектах `Vector`, составленных из префиксов и суффиксов. Можно воспользоваться хэш-таблицей типа `Hashtable`, содержащей префиксы в качестве ключей и векторы суффиксов в качестве значений. Для конструкции такого типа существует термин *таблица соответствий* — в данном случае соответствий между префиксами и суффиксами. В Java нам не придется организовывать явный тип `State` для состояния, поскольку в объекте `Hashtable` префиксы и суффиксы уже связаны неявным обра-

зом (т.е. между ними установлено соответствие). Все это сильно отличается от реализации на С, в которой организовывались структуры `State`, содержащие префиксы и списки суффиксов, а затем по префиксу строился хэш-код для получения полного состояния (`State`).

Класс `Hashtable` содержит метод `put` для помещения в таблицу пары "ключ–значение", а также метод `get` для извлечения значения по ключу:

```
Hashtable h = new Hashtable();
h.put(key, value);
Sometype v = (Sometype) h.get(key);
```

Наша реализация включает три класса. Первый класс, `Prefix`, содержит слова префикса:

```
class Prefix {
    public Vector pref; // NPREF соседних слов из входного потока
    ...
```

Второй класс, `Chain`, считывает входные данные, строит хэш-таблицу и генерирует текст. Вот его переменные-члены:

```
class Chain {
    static final int NPREF = 2; // длина префикса
    static final String NONWORD = "\n";
                 // фиктивное "слово", которого нет в тексте
    Hashtable statetab = new Hashtable();
                 // ключ - префикс, значение - вектор суффиксов
    Prefix prefix = new Prefix(NPREF, NONWORD);
                 // начальный префикс
    Random rand = new Random();
    ...
```

Третий класс представляет собой открытый интерфейс, который содержит функцию `main` и создает экземпляр класса `Chain`:

```
class Markov {
    static final int MAXGEN = 10000; // максимум слов на выходе
    public static void main(String[] args) throws IOException
    {
        Chain chain = new Chain();
        int nwords = MAXGEN;

        chain.build(System.in);
        chain.generate(nwords);
    }
}
```

При создании экземпляра класса `Chain` он, в свою очередь, создает хэш-таблицу и помещает в нее начальные префиксы в виде `NPREF` элементов `NONWORD`. Функция `build` вызывает библиотечную функцию `StreamTokenizer` для разбора входного потока на отдельные слова, разделенные пустым пространством. Три вызова перед началом цикла переводят лексический анализатор в нужное состояние согласно нашему определению "слова".

```
// Chain.build: строит из входного потока таблицу State
void build(InputStream in) throws IOException
{
    StreamTokenizer st = new StreamTokenizer(in);

    st.resetSyntax();          // отмена стандартных правил
    st.wordChars(0, Character.MAX_VALUE); // вкл. все символы
    st.whitespaceChars(0, ' ');           // кроме пробела и т.п.
    while (st.nextToken() != st.TT_EOF)
        add(st.sval);
    add(NONWORD);
}
```

Функция add извлекает вектор суффиксов, соответствующий текущему префиксу, из хэш-таблицы. Если таковых нет (вектор пуст), функция add создает новый вектор и новый префикс для помещения в таблицу. В любом случае к вектору суффиксов добавляется новое слово, а затем префикс передвигается вперед путем отбрасывания первого слова и добавления нового в его конец.

```
// Chain.add: добавляет слово в список суффиксов префикса
void add(String word)
{
    Vector suf = (Vector) statetab.get(prefix);
    if (suf == null) {
        suf = new Vector();
        statetab.put(new Prefix(prefix), suf);
    }
    suf.addElement(word);
    prefix.pref.removeElementAt(0);
    prefix.pref.addElement(word);
}
```

Заметьте, что если объект suf — нулевой, то функция add помещает в хэш-таблицу новый объект Prefix, а не сам prefix. Это необходимо по той причине, что в хэш-таблицах типа Hashtable элементы хранятся по ссылке, а не по значению, так что если не сделать копию, можно затереть предыдущие данные таблицы. С этим обстоятельством мы уже сталкивались при реализации алгоритма на языке C.

Функция генерирования текста аналогична C-версии, но несколько компактнее, поскольку прямой доступ к элементам вектора быстрее, чем последовательный перебор списка.

```
// Chain.generate: генерирование текста на выходе
void generate(int nwords)
{
    prefix = new Prefix(NPREF, NONWORD);
    for (int i = 0; i < nwords; i++) {
        Vector s = (Vector) statetab.get(prefix);
        int r = Math.abs(rand.nextInt()) % s.size();
        String suf = (String) s.elementAt(r);
        if (suf.equals(NONWORD))
            break;
        System.out.println(suf);
        prefix.pref.removeElementAt(0);
```

```
            prefix.pref.addElement(suf);
        }
    }
```

Два конструктора класса `Prefix` создают два экземпляра на основе исходных данных. Первый из них копирует существующий объект `Prefix`, а второй создает префикс из `n` копий строки; именно второй конструктор используется для создания `NPERF` копий строки `NONWORD` при инициализации:

```
// Конструктор Prefix: копирует существующий префикс
Prefix(Prefix p)
{
    pref = (Vector) p.pref.clone();
}

// Конструктор Prefix: создает n копий str
Prefix(int n, String str)
{
    pref = new Vector();
    for (int i = 0; i < n; i++)
        pref.addElement(str);
}
```

В классе `Prefix` есть еще два метода, `hashCode` и `equals`, вызываемые неявно из реализации класса `Hashtable` для индексирования и поиска в таблице. Именно необходимость иметь класс с реализацией этих двух методов для корректной работы объекта `Hashtable` заставила нас сделать `Prefix` полноценным самостоятельным классом, а не просто объектом `Vector`, как суффиксы.

Метод `hashCode` строит единый хэш-код, объединяя набор хэш-кодов для элементов вектора:

```
static final int MULTIPLIER = 31;  // для hashCode()

// Prefix.hashCode: генерирует хэш-код из всех слов префикса
public int hashCode()
{
    int h = 0;

    for (int i = 0; i < pref.size(); i++)
        h = MULTIPLIER * h + pref.elementAt(i).hashCode();
    return h;
}
```

Метод `equals` выполняет поэлементное сравнение слов в двух префиксах:

```
// Prefix.equals: сравнение двух префиксов
public boolean equals(Object o)
{
    Prefix p = (Prefix) o;

    for (int i = 0; i < pref.size(); i++)
        if (!pref.elementAt(i).equals(p.pref.elementAt(i)))
            return false;
    return true;
}
```

Программа на Java существенно короче, чем на C, поскольку в ней многие операции выполняются автоматически стандартными средствами. Очевидными примерами могут служить классы `Vector` и `Hashtable`. В целом управление памятью устроено несложно, потому что векторы растут по мере необходимости, а сборщик мусора заботится об освобождении и возвращении в систему той памяти, к которой больше не обращаются. Но для корректного использования класса `Hashtable` необходимо написать функции `hashCode` и `equals`, поэтому нельзя так уж однозначно сказать, что Java заботится обо всех деталях реализации.

Сравнивая представление и способы работы с одной и той же абстрактной структурой данных в C и Java, сразу можно заметить, что в Java обеспечивается лучшее разделение функций. Например, перейти от объектов `Vector` к массивам было бы совсем просто. В C-версии каждый модуль “знает”, что делают все остальные: хэш-таблица работает с массивами, которые хранятся в другом месте, функции `lookup` известно устройство структур `State` и `Suffix`, и длина массива префиксов также является всеобщим достоянием.

Ниже приведена командная строка для выполнения программы с одним из текстовых файлов:

```
java Markov <jr_chemistry.txt | fmt
```

А вот результат работы:

```
Wash the blackboard.  Watch it dry.  The water goes
into the air.  When water goes into the air it
evaporates. Tie a damp cloth to one end of a solid or liquid.
Look around.  What are the solid things?
Chemical changes take place when something burns. If
the burning material has liquids, they are stable and
the sponge rise. It looked like dough, but it is
burning. Break up the lump of sugar into small pieces
and put them together again in the bottom of a liquid.
(Вымойте доску. Посмотрите ее досуха. Вода поднимается в воздух.
Когда вода поднимается в воздух, она испаряется. Привяжите мокрую
ткань к одному концу твердого тела или жидкости. Оглянитесь
вокруг. Что такое твердые тела? Когда что-то горит, происходят
химические изменения. Если горящий материал содержит жидкости,
они устойчивы и губка повышаются. Это было похоже на тесто, но
оно горит. Расколите кусок сахара на мелкие кусочки и снова
соберите их у дна жидкости.)
```

Упражнение 3.4. Измените Java-версию марковского алгоритма так, чтобы для хранения префиксов в классе `State` использовался массив, а не объект `Vector`.

3.6. C++

Третью реализацию того же алгоритма мы напишем на C++. Поскольку C++ практически представляет собой надмножество C, писать на нем можно так же, как и на C, с некоторыми дополнительными удобствами. Первая версия программы `markov` на языке C одновременно является и программой на C++. И все же для C++ более характерно определение классов для объектов, используемых в программе, и

очень похоже на то, как это делалось в Java. Таким образом можно скрыть подробности реализации. Мы решили зайти еще дальше и воспользоваться *стандартной библиотекой шаблонов* (Standard Template Library), или STL, поскольку в этой библиотеке имеются встроенные механизмы, делающие многое из того, что нам нужно. Стандарт ISO языка C++ рассматривает STL как часть определения языка.

Библиотека STL содержит такие контейнеры, как векторы, списки и множества, а также семейство фундаментальных алгоритмов для поиска, сортировки, добавления и удаления элементов. С использованием средств работы с шаблонами в языке C++ все алгоритмы STL могут применяться к самым разным контейнерам, в том числе к нестандартным структурным типам или элементарным встроенным, наподобие целочисленного. Контейнеры определяются в форме шаблонов C++, экземпляры которых создаются в нужное время для того или иного типа помещаемых данных. Например, имеется шаблон контейнера `vector`, из которого можно образовывать конкретные типы наподобие `vector<int>` или `vector<string>`. Все операции над шаблоном `vector`, в том числе стандартные алгоритмы сортировки, доступны для работы с вновь определенными типами данных.

Кроме контейнера `vector`, напоминающего тип `Vector` из Java, библиотека STL содержит еще и контейнер `deque`. Этот тип данных (его название произносится как "дек" и происходит от "double-ended queue") представляет собой двустороннюю очередь, выполняющую такие операции, какие мы проделывали с префиксами: содержит `NPREF` элементов, позволяет извлекать первый элемент из начала и добавлять новый в конец, и время обеих операций составляет $O(1)$. Тип `deque` из библиотеки STL имеет несколько более общий характер, чем необходимо, поскольку позволяет извлекать и помещать данные с обоих концов, но гарантия его быстродействия дает нам основание решительно высказаться в его пользу.

В STL имеется также контейнер `map`, базирующийся на использовании сбалансированных деревьев и содержащий пары "ключ–значение". Элементы, ассоциированные с теми или иными ключами, извлекаются из него за время порядка $O(\log n)$. Эти таблицы соответствий могут работать не так быстро, как таблицы с порядком быстродействия $O(1)$, но зато привлекают перспективой вообще не дописывать никакого кода. (В некоторых нестандартных библиотеках C++ имеются контейнеры `hash` или `hash_map` с еще более высоким быстродействием.)

Мы также пользуемся встроенными функциями сравнения, которые в данном случае сравнивают отдельные строки в префиксах.

Имея под рукой все перечисленные компоненты, из них уже легко собрать нужную программу. Вот необходимые объявления:

```
typedef deque<string> Prefix;
map<Prefix, vector<string> > statetab; // префикс -> суффиксы
```

В библиотеке STL имеется шаблон для двусторонних очередей; употребление его в виде `deque<string>` создает очередь со строковыми элементами. Поскольку этот тип фигурирует в программе несколько раз, мы дали ему имя `Prefix` с помощью оператора `typedef`. Однако таблица соответствий, в которой хранятся префиксы и суффиксы, встречается всего один раз, так что давать ее типу отдельное имя было незачем. Объявляется переменная `statetab` типа `map`, представляющая собой таблицу соот-

ветствий между префиксами и векторами строк. Это удобнее, чем в C и Java, поскольку не нужно дополнительно писать ни функции hash, ни метод equals.

Главный модуль программы инициализирует префикс, считывает входные данные (из стандартного потока ввода, носящего имя cin в библиотеке iostream языка C++), добавляет "хвост" и генерирует текст таким же образом, как и в предыдущих версиях.

```
// Главный модуль: генерирует случайный текст
int main(void)
{
    int  nwords = MAXGEN;
    Prefix prefix;                  // текущий префикс

    for (int i = 0; i < NPREF; i++) // инициализация префиксов
        add(prefix, NONWORD);
    build(prefix, cin);
    add(prefix, NONWORD);
    generate(nwords);
    return 0;
}
```

В функции build с помощью средств библиотеки iostream входной поток считывается слово за словом:

```
// build: считывает данные, строит таблицу состояний
void build(Prefix& prefix, istream& in)
{
    string buf;

    while (in >> buf)
        add(prefix, buf);
}
```

Строка buf расширяется по мере необходимости для работы с поступающими на вход словами произвольной длины.

Функция add демонстрирует дополнительные преимущества работы с библиотекой STL:

```
// add: добавляет слово в список суффиксов, обновляет префикс
void add(Prefix& prefix, const string& s)
{
    if (prefix.size() == NPREF) {
        statetab[prefix].push_back(s);
        prefix.pop_front();
    }
    prefix.push_back(s);
}
```

За этими внешне простыми операторами скрывается огромный объем проделываемой работы. Контейнер map перегружает операцию обращения по индексу (знак [ ]) так, чтобы она выполнялась как операция поиска по таблице. Выражение statetab[prefix] служит для выполнения поиска в таблице statetab по ключу prefix; оно возвращает ссылку на требуемую позицию. Если вектор не существует, он создается. Функции-методы с именем push_back классов vector и deque

помещают новую строку в конец вектора или очереди; метод `pop_front` извлекает первый элемент из очереди.

Генерирование текста выполняется аналогично предыдущим версиям:

```
// generate: вывод текста по одному слову в строке
void generate(int nwords)
{
    Prefix prefix;
    int i;

    for (i = 0; i < NPREF; i++) // инициализация
        add(prefix, NONWORD);

    for (i = 0; i < nwords; i++) {
        vector<string>& suf = statetab[prefix];
        const string& w = suf[rand() % suf.size()];
        if (w == NONWORD)
            break;
        cout << w << "\n";
        prefix.pop_front();  // сдвиг вперед
        prefix.push_back(w);
    }
}
```

В целом именно эта версия выглядит особенно четко и элегантно. Код компактен, структура данных четко определена, алгоритм совершенно очевиден. К сожалению, за все приходится платить: данная версия работает значительно медленнее, чем первоначальная программа на С, хотя она и не самая медленная. Через некоторое время мы еще вернемся к вопросам сравнения быстродействия.

Упражнение 3.5. К сильным сторонам библиотеки STL относится та легкость, с которой ее пользователь может экспериментировать с различными структурами данным. Измените приведенную выше программу на С++ так, чтобы для представления префиксов, списков суффиксов и таблицы состояний в ней использовались другие структуры данных. Как эти изменения влияют на быстродействие?

Упражнение 3.6. Напишите такую версию программы на С++, в которой бы использовались только классы и тип `string`, но не применялось бы ни одно из библиотечных средств STL. Сравните ее по стилю и быстродействию с STL-версией.

3.7. Awk и Perl

В завершение нашего упражнения мы написали версии данной программы также на двух популярных языках разработки сценариев — Awk и Perl. В них имеются все необходимые средства для решения этой задачи, такие как ассоциативные массивы и методы обработки строк.

Ассоциативный массив — это удобный способ представления хэш-таблицы. Он очень похож на массив, только его индексами могут быть произвольные числа, или строки, или даже списки таких элементов, разделенных запятыми. Это своеобразная форма установки соответствия между двумя типами данных. В языке Awk

все массивы — ассоциативные, а в Perl имеются как обычные массивы с целочисленными индексами, так и ассоциативные массивы (они называются *хэш-массивами*, что однозначно указывает на способ их реализации).

Версии нашей программы на Awk и Perl узко специализированы для работы с префиксами длины 2.

```
# markov.awk: марковский алгоритм с префиксами длиной 2
BEGIN {  MAXGEN = 10000; NONWORD = "\n"; w1 = w2 = NONWORD }

{   for (i = 1; i <= NF; i++) {   # считываются все слова
        statetab[w1,w2,++nsuffix[w1,w2]] = $i
        w1 = w2
        w2 = $i
    }
}
END {
    statetab[w1,w2,++nsuffix[w1,w2]] = NONWORD  # хвост
    w1 = w2 = NONWORD
    for (i = 0; i < MAXGEN; i++) {  # генерируем
         r = int(rand()*nsuffix[w1,w2]) + 1  # nsuffix >= 1
         p = statetab[w1,w2,r]
         if (p == NONWORD)
             exit
         print p
         w1 = w2       # сдвиг по цепочке
         w2 = p
    }
}
```

Awk представляет собой язык для работы с шаблонами и образцами. Из входных данных считывается одна строка, затем эта строка сравнивается с заданными образцами, и для каждого найденного совпадения выполняются определенные действия. Имеются два особых шаблона, `BEGIN` и `END`, которые обнаруживаются программой перед первой строкой данных и после последней.

Упомянутые "определенные действия" — это блоки операторов в фигурных скобках. В Awk-версии нашей программы блок `BEGIN` инициализирует префикс и еще пару переменных.

Следующий блок не соответствует никакому образцу, поэтому по умолчанию выполняется один раз для каждой поступающей строки входных данных. Awk автоматически разбивает каждую прочитанную строку на поля (слова, ограниченные символами пустого пространства) с именами от `$1` до `$NF`; переменная `NF` обозначает количество полей. Следующий оператор строит таблицу соответствий между префиксами и суффиксами:

```
statetab[w1,w2,++nsuffix[w1,w2]] = $i
```

В массиве `nsuffix` подсчитываются суффиксы, а в элементе `nsuffix[w1,w2]` — количество суффиксов, ассоциированных с данным префиксом. Сами суффиксы хранятся в элементах массива `statetab[w1,w2,1]`, `statetab[w1,w2,2]` и т.д.

Блок `END` выполняется тогда, когда все входные данные прочитаны. К этому моменту с каждым префиксом ассоциируется элемент массива `nsuffix`, содержа-

щий количество соответствующих префиксу суффиксов, и ровно столько элементов массива statetab, содержащих сами суффиксы.

Версия программы на языке Perl устроена похожим образом, но в ней для подсчета суффиксов используется не третий индекс, а анонимный массив. Для обновления префикса применяется множественное присваивание. В Perl типы переменных обозначаются специальными символами: $ соответствует скаляру, а @ — индексированному массиву, причем квадратные скобки используются для индексирования массивов, а фигурные — для индексирования хэш-массивов.

```
# markov.pl: марковский алгоритм с префиксами длиной 2

$MAXGEN = 10000;
$NONWORD = "\n";
$w1 = $w2 = $NONWORD;              # начальное состояние
while (<>) {                       # считывание данных по строке
    foreach (split) {
        push(@{$statetab{$w1}{$w2}}, $_);
        ($w1, $w2) = ($w2, $_); # множественное присваивание
    }
}
push(@{$statetab{$w1}{$w2}}, $NONWORD);   # хвост

$w1 = $w2 = $NONWORD;
for ($i = 0; $i < $MAXGEN; $i++) {
    $suf = $statetab{$w1}{$w2}; # обращение к массиву
    $r = int(rand @$suf);          # @$suf - количество элементов
    exit if (($t = $suf->[$r]) eq $NONWORD);
    print "$t\n";
    ($w1, $w2) = ($w2, $t);        # сдвиг по цепи
}
```

Как и в предыдущих программах, таблица соответствий хранится в переменной statetab. Самым сердцем программы является следующая строка:

```
push(@{$statetab{$w1}{$w2}}, $_);
```

Этот оператор помещает новый суффикс в конец анонимного массива, хранящегося по адресу statetab{$w1}{$w2}. На этапе генерирования выражение $statetab{$w1}{$w2} является ссылкой на массив суффиксов, а $suf->[$r] указывает на *r*-й суффикс по порядку.

Программы на Awk и Perl оказались значительно короче своих предшественниц, написанных на трех других языках. Но зато их намного труднее адаптировать для обработки префиксов длины, отличной от двух. Ядро версии на C++ с применением средств STL (функции add и generate) имеет сравнимую длину, однако устроено гораздо понятнее. И все же именно языки разработки сценариев часто оказываются особенно удачными для программных экспериментов, для написания пробных прототипов и даже для разработки профессиональных версий программ, если быстродействие не является ключевым требованием.

Упражнение 3.7. Измените версии программы на языках Awk и Perl так, чтобы они смогли обрабатывать префиксы любой длины. Определите экспериментальным путем, как это влияет на быстродействие.

3.8. Вопросы быстродействия

Итак, у нас для сравнения есть несколько программных реализаций одного и того же алгоритма. Мы испытали эти программы на Псалтыри (книге "Псалмов Давидовых") из англоязычной Библии в канонической редакции короля Иакова. Английский текст Псалтыри содержит 42 685 слов (из них 5 238 различных) или 22 482 префиксов. В этом тексте достаточно много повторяющихся словосочетаний наподобие "Blessed is the..." ("Блажен..."), так что один из списков суффиксов содержит более 400 элементов, и есть еще несколько сотен цепочек с десятками суффиксов. Поэтому данный текст является хорошим тестовым примером.

```
Blessed is the man of the net. Turn thee unto me, and raise me up,
that I may tell all my fears. They looked unto him, he heard. My
praise shall be blessed. Wealth and riches shall be saved. Thou hast
dealt well with thy hid treasure; they are cast into a standing water,
the flint into a standing water, and dry ground into watersprings.
(Блажен муж из сети. Обратись на меня и вознеси меня, дабы поведал я
все свои страхи. Воззрились они на него, и он услышал. Благословенна
будь хвала моя. Богатство и богатеи да спасутся. Ты хорошо поступил со
скрытым сокровищем; они брошены в тихую воду, кремень в тихую воду, и
землю иссохшую в источники вод.)
```

Ниже приведена таблица с данными о времени, которое потребовалось для генерирования 10 000 слов выходного текста. Эксперименты проводились на компьютерах MIRS R10000 с тактовой частотой 250 МГц под управлением операционной системы Irix 6.4 и Pentium II (тактовая частота 400 МГц, 128 Мбайт памяти) под управлением Windows NT. Время выполнения задачи почти целиком определяется объемом входных данных; генерирование в сравнении с ним происходит очень быстро. Таблица содержит также примерные длины программ, измеренные в строках исходного кода.

	250 МГц, R10000	*400 МГц, Pentium II*	*Строк в исходном коде*
C	0,36 с	0,30 с	150
Java	4,9	9,2	105
C++/STL/очередь	2,6	11,2	70
C++/STL/список	1,7	1,5	70
Awk	2,2	2,1	20
Perl	1,8	1,0	18

Версии на языках C и C++ компилировались оптимизирующими компиляторами, а при запуске программы на Java разрешалась компиляция по требованию. Время выполнения программ на C и C++ в системе Irix было отобрано как лучший результат из числа выданных тремя разными компиляторами. Подобные же результаты наблюдались на машинах Sun SPARC и DEC Alpha. C-версия этой программы быстрее остальных во много раз. Второе место по быстродействию занимает Perl. И все же данные, представленные в нашей таблице, получены в частном случае для определенного набора компиляторов и библиотек, поэтому в другой среде экспериментальные результаты могут сильно отличаться.

Совершенно очевидно, что версия с использованием очередей (deque) из библиотеки STL в системе Windows содержит какой-то принципиальный недостаток. Эксперименты показали, что именно двусторонняя очередь для представления префиксов виновата в потере машинного времени, хотя она и не содержит более двух элементов. Скорее мы ожидали, что основным виновником окажется центральная структура данных — таблица соответствий. Переход от очереди к списку (в STL это двусвязный или двунаправленный список) радикально улучшает быстродействие. С другой стороны, переключение с таблицы на нестандартный контейнер hash оказалось вообще незаметным в системе Irix. В системе Windows такая структура данных была недоступна. То, что структуры данных в STL фундаментально продуманы и спроектированы с высоким качеством, легко видеть из того факта, что подобные переходы потребовали всего лишь подстановки слова list вместо слова deque или hash вместо map в двух местах исходного кода и перекомпиляции программы. Остается заключить, что библиотека STL, новейший компонент языка C++, все еще грешит несовершенствами реализации. Соотношение быстродействия между программами со специально разработанными структурами данных и программами со структурами из STL совершенно непредсказуемо. То же самое верно и в отношении языка Java, в котором различные реализации сменяют друг друга очень быстро.

Здесь имеются интересные направления для дальнейшего экспериментирования, например, тестирование программы с большим запрашиваемым объемом текста на выходе. Откуда нам знать, работает ли она вообще? Как убедиться в том, что программа часть своего времени не теряет впустую? В главе 6, посвященной вопросам тестирования, даны некоторые рекомендации, а также описано, как мы тестировали программу с марковским алгоритмом.

3.9. Извлекаем уроки

Программа работы с текстом на основе цепей Маркова имеет долгую историю. Первую версию написал Дон Митчелл (Don P. Mitchell), адаптировал Брюс Эллис (Bruce Ellis), и в 1980-х годах она использовалась для различных курьезных демонстраций, связанных с перетасовкой данных. Эта программа лежала без дела до тех пор, пока мы не решили использовать ее в университетском курсе для иллюстрации процесса разработки программ. Вместо того чтобы сдувать пыль с оригинала, мы написали программу заново на языке С, по ходу освежая воспоминания о возникающих при этом проблемах, а затем перевели еще на несколько языков, стараясь выразить одну и ту же идею по-разному идиоматическими средствами каждого языка. После окончания курса мы еще многократно переработали эти программы, постепенно улучшая четкость их структуры и наглядность представления операций.

Тем не менее все это время основной подход, костяк алгоритма, оставался тем же самым. Более ранняя версия реализовала тот же подход, что и представленная в этой главе, только в ней использовалась вторая хэш-таблица, содержащая отдельные слова. Если бы нужно было переписать программу еще раз, мы бы вряд ли стали что-то менять. Структура программы вырастает из организации ее данных. Структуры данных, конечно, не определяют реализацию программы вплоть до мелких подробностей, но несомненно придают ей общую форму.

Выбор между некоторыми структурами данных оказывается непринципиальным, например, отличие между списком и расширяемым массивом в данной программе несущественно. В некоторых реализациях степень общности алгоритма выше или ниже, чем в других. Так, в программах на Awk и Perl можно довольно просто перейти к префиксам из одного или, скажем, трех слов, а вот параметризация для работы с префиксами произвольной длины уже представляет собой проблему. Как и подобает объектно-ориентированным языкам, в C++ и Java требуются совсем крошечные изменения в программах, чтобы приспособить структуры данных к обработке другой информации, отличной от литературного английского текста. Это могут быть, например, исходные тексты программ (где символы пустого пространства играют большую роль), или музыкальные ноты, или даже щелчки мышью и выбор пунктов меню, позволяющие генерировать текстовые последовательности.

Разумеется, хотя структуры данных остаются более-менее теми же самыми, возможны большие вариации в общем строении программ, длине их исходного кода и быстродействии. Грубо говоря, языки высокого уровня порождают более медленный код, чем языки низкого уровня; конечно, несмотря на это, мы неизменно стремимся к качественному обобщению. Большие структурные единицы программ, такие как библиотечные средства STL в C++ или ассоциативные массивы и методы работы со строками в языках разработки сценариев, позволяют добиться компактности исходного кода и выполнить работу в кратчайшие сроки. За это приходится расплачиваться падением скорости выполнения, хотя быстродействие и не всегда играет большую роль в таких программах, которые выполняются всего несколько секунд.

Менее очевидным представляется вопрос о том, как же оценить степень потери контроля над программой в том случае, когда системные и библиотечные средства, нагромождаясь друг на друга, образуют такую кучу, в которой уже непонятно, что происходит на нижнем уровне организации. Именно так обстоят дела с STL-версией программы — ее быстродействие непредсказуемо, и простого способа исправить ситуацию не существует. Одна из недоработанных реализаций этой библиотеки потребовала серьезной переработки, прежде чем она вообще смогла обеспечить выполнение нашей программы. Лишь у немногих из нас хватает энергии проследить такую проблему до конца и разрешить ее.

В индустрии разработки программного обеспечения существует давняя, повсеместная и все усугубляющаяся проблема: библиотеки, интерфейсы и инструментальные средства становятся все сложнее, и по мере этого уровень понимания и способность ими управлять сильно падают. Когда все работает как следует, развитые среды программирования позволяют добиться очень высокой производительности труда, но если вдруг возникает сбой, то деваться некуда. Можно вообще не понять, что проблема существует и где ее источник, если речь идет о необъяснимом падении быстродействия или тонких ошибках в управляющих конструкциях.

Проектирование и реализация приведенных программ позволяет извлечь ряд уроков на будущее, для целей разработки более сложных и длинных программ. Во-первых, важно вначале выбрать простые алгоритмы и структуры данных — простейшие из тех, что способны выполнить работу ожидаемого объема за разумное время. Если такие средства уже кем-то написаны и объединены в библиотеку, тем лучше — именно с использованием подобных средств мы и написали нашу версию на C++.

Следуя принципу, приведенному в эпиграфе, мы начали с тщательного конструирования структур данных, имея в виду алгоритм, который будет к ним применяться. Подготовив структуры данных, мы легко собрали код воедино.

Трудно спроектировать программу полностью и сразу же реализовать ее; при написании реальных программ неизбежен процесс последовательных приближений и экспериментирования. Написание кода требует от программиста четкой проработки тех конструкций и технических решений, в отношении которых на этапе замысла еще можно было обойтись натяжками и прикидками. Именно так и обстояло дело с нашими программами из этой главы, которые подверглись неоднократной доработке и переработке. По возможности всегда начинайте с чего-то самого простого и двигайтесь вперед, как вам подсказывает опыт. Если бы наша цель заключалась только в том, чтобы написать версию марковского алгоритма для личного пользования и развлечения, мы бы почти наверняка обошлись вариантом на Awk или Perl (причем не столь тщательно проработанным), не вдаваясь в тонкости.

Однако коммерческая или профессиональная версия программы всегда отнимает больше времени и усилий, чем ее технический прототип. Если бы программы, рассмотренные в этой главе, пришлось пускать в профессиональную работу, это было бы возможно — они всесторонне протестированы и выверены по стилю. Достижение профессионального качества требует на порядок или два больше трудозатрат, чем разработка программы для личного пользования.

Упражнение 3.8. Мы знакомы с версиями нашей программы реализации марковских цепей на целом ряде языков: Scheme, Tcl, Prolog, Python, Generic Java, ML и Haskell; каждый из них имеет свои преимущества и проблемные места. Напишите эту же программу на одном из ваших любимых языков и сравните ее стиль и быстродействие с существующими.

Дополнительная литература

Стандартная библиотека шаблонов STL описана во многих книгах, среди которых: Matthew Austern, *Generic Programming and the STL* (Addison-Wesley, 1998). В качестве справочника по языку C++, несомненно, стоит рекомендовать книгу Bjarne Stroustrup, *The C++ Programming Language* (3rd edition, Addison-Wesley, 1997). Что касается Java, можно воспользоваться книгой Ken Arnold, James Gosling, *The Java Programming Language, 2nd Edition* (Addison-Wesley, 1998). Лучшее описание языка Perl дано в книге Larry Wall, Tom Christiansen, Randal Schwartz, *Programming Perl, 2nd Edition* (O'Reilly, 1996).

Существует идея *структурных шаблонов* (*design patterns*), заключающаяся в том, что все программы строятся на основе лишь нескольких фундаментальных организующих конструкций, точно так же, как все задачи реализуются на основе комбинаций из нескольких базовых структур данных. Очень приблизительно можно сказать, что это аналоги идиоматических программных конструкций, рассмотренных в главе 1. Общепринятым справочником по этим вопросам является книга Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

Невероятные приключения программы `markov`, первоначально существовавшей под именем `shaney`, были описаны в рубрике "Computing Recreations" журнала *Scientific American* за июнь 1989 года. Эта же статья была перепечатана в книге A.K.Dewdney, *The Magic Machine* (W. H. Freeman, 1990).

4

Интерфейсы

Уж раз мы строим стену, надо знать,
Что защитит она, и от кого,
И в чем кому-то от меня обида.
Есть нелюбовь какая-то к стене,
Мечта ее обрушить!

Роберт Фрост. "Починка стены"[1]
(Robert Frost, Mending Wall*)*

В процессе разработки программ приходится постоянно балансировать между трудно совместимыми целями и удовлетворять противоречивым ограничениям. Даже при написании небольшой автономной системы приходится идти на многие компромиссы, но там сделанный выбор по крайней мере остается спрятанным внутри и оказывает влияние только на работу самого автора программы. А вот если код необходимо передавать для использования другим людям, то сделанные выборы и компромиссы будут иметь гораздо более отдаленные последствия.

Среди вопросов, которые необходимо проработать при проектировании программной системы, имеются следующие.

- Интерфейсы: какие услуги и какой доступ к ним необходимо предоставить? Интерфейс — это по сути договор между поставщиком и заказчиком. Обычно стремятся обеспечить удобный и единообразный набор услуг, чтобы функциональные возможности были под рукой в достаточном количестве, но не слишком загромождали оперативное пространство.
- Сокрытие информации: какие данные должны быть видимы пользователю, а какие — скрыты от него? Интерфейс должен обеспечивать несложный доступ к компонентам и при этом скрывать детали реализации так, чтобы ее можно было дорабатывать незаметно для пользователя.
- Управление ресурсами: кто должен отвечать за распределение памяти и других ограниченных ресурсов? Основные проблемы — это как размещать и удалять объекты из памяти и как распоряжаться совместно используемыми экземплярами данных.
- Обработка ошибок: как распознавать ошибки и выдавать сообщения о них? Какие меры необходимо принять, если произошла та или иная ошибка?

---

[1] Перевод А. Цветкова

В главе 2 рассматривались элементарные “кирпичики” — структуры данных, из которых строятся программы. В главе 3 изучался вопрос, как объединить такие “кирпичики” в одну небольшую программу. Теперь же нашей темой будет организация интерфейсов между компонентами, которые, вполне возможно, происходят из разных источников. В этой главе мы проиллюстрируем проектирование интерфейса на примере разработки библиотеки функций и структур данных для одной распространенной задачи. Попутно будут рассмотрены некоторые принципы проектирования программ. Как правило, в такой работе приходится принимать невероятное количество решений, но большинство из них — почти бессознательно. Не имея на вооружении нужных принципов, можно лишь нагромождать бессистемные конструкции, постоянно мешающие программистам в работе и раздражающие их.

4.1. Данные, разделенные запятыми

Данные, разделенные запятыми (*Comma-Separated Values*, или *CSV*), — это термин для естественного и часто встречающегося способа представления табличных данных. Каждая строка таблицы представляет собой строку текста; поля в каждой строке отделяются друг от друга запятыми. Например, английский оригинал таблицы, приведенной в конце предыдущей главы, мог бы начинаться в этом формате следующим образом:

```
,"250MHz","400MHz","Lines of"
,"R10000","Pentium II","source code"
C,0.36 sec,0.30 sec,150
Java,4.9,9.2,105
```

В таком формате данные считываются и записываются программами работы с электронными таблицами. Не случайно именно в этом формате они фигурируют и на Web-страницах, посвященных услугам типа биржевых котировок. Одна из популярных Web-страниц биржевых котировок представляет информацию примерно в следующем виде.

Обозначение	Последняя сделка		Изменение		Объем
LU	2:19PM	86-1/4	+4-1/16	+4,94%	5,804,800
T	2:19PM	60-11/16	-1-3/16	-1,92%	2,468,000
MSFT	2:24PM	106-9/16	+1-3/8	+1.31%	11,474,900

Скачать в формате электронной таблицы

Считывать цифры из окна Web-браузера довольно удобно, однако это отнимает много времени. Представьте себе всю эту волокиту: запустить браузер, подождать, прорваться через рекламный мусор на экране, ввести интересующие вас наименования, снова подождать (на этот раз еще дольше), снова терпеливо снести рекламу, и все ради нескольких цифр. К тому же дальнейшая обработка этих цифр требует дополнительных операций. После щелчка на ссылке “Скачать в формате электронной таблицы” откроется файл, содержащий ту же информацию в виде строк данных, разделенных запятыми (здесь эти данные отредактированы, чтобы помещаться в строки):

```
"LU",86.25,"11/4/1998","2:19PM",+4.0625,
        83.9375,86.875,83.625,5804800
"T",60.6875,"11/4/1998","2:19PM",-1.1875,
        62.375,62.625,60.4375,2468000
"MSFT",106.5625,"11/4/1998","2:24PM",+1.375,
        105.8125,107.3125,105.5625,11474900
```

Что во всей этой процедуре мозолит глаза, так это отсутствие главного преимущества компьютеризации — автоматической обработки данных. С помощью браузера можно обращаться к данным на сетевом сервере, но было бы еще удобнее загружать их на свой компьютер автоматически, без принудительного вмешательства. За всеми этими щелчками на кнопках стоит чисто текстуальная процедура: браузер считывает HTML-код, вы вводите текст, браузер посылает его на сервер и снова считывает оттуда HTML-код. Имея под рукой нужные средства и соответствующий язык, можно легко получить эту информацию автоматически. Ниже приведена программа на языке Tcl для обращения к Web-сайту биржевых котировок и загрузки данных в показанном выше CSV-формате с несколькими строками заголовка, предваряющими сами данные.

```
# getquotes.tcl: биржевые котировки Lucent, AT&T и Microsoft

set so [socket quote.yahoo.com 80]  ;# соединение
set q "/d/quotes.csv?s=LU+T+MSFT&f=sl1d1t1c1ohgv"

puts $so "GET $q HTTP/1.0\r\n\r\n"  ;# запрос
flush $so
puts [read $so]                     ;# чтение и ответ
```

Загадочная строка `f=...` после символов акций представляет собой недокументированную управляющую строку, аналогичную первому строковому аргументу функции `printf`. Она определяет, какие данные нужно загрузить. Путем экспериментирования мы определили, что `s` обозначает символ акции, `l1` — последнюю цену, `c1` — изменение цены со вчерашнего дня и т.д. Но важны не столько сами детали, которые все равно со временем изменяются, сколько возможность автоматизации: получение требуемой информации и преобразование ее в нужную форму без вмешательства человека. Таким образом, машина сама делает всю работу.

Для выполнения программы `getquotes` обычно требуется доля секунды, т.е. намного меньше, чем для работы с браузером. Получив данные, необходимо дополнительно обработать их. Такие форматы, как CSV, удобны для работы в том случае, если имеются библиотеки для преобразования данных из этого формата и в него, а также для некоторых дополнительных операций наподобие преобразований чисел. Но нам неизвестна ни одна библиотека открытого пользования для работы с форматом CSV, поэтому придется написать ее самостоятельно.

В следующих разделах мы создадим три версии библиотеки для считывания данных, разделенных запятыми, и преобразования их в международные представления. Заодно мы поговорим о проблемах, которые возникают при разработке программ, взаимодействующих с другими программами. Например, не существует стандартного определения формата CSV, поэтому реализация библиотеки не может основываться на точной спецификации. Это весьма распространенная ситуация при разработке интерфейсов.

4.2. Прототип библиотеки

Маловероятно, чтобы удалось правильно спроектировать библиотеку или интерфейс с первой же попытки. Как однажды написал Фредерик Брукс (Frederick Brooks): "...заранее приготовьтесь выбросить ваше творение; все равно придется". Брукс писал о больших системах, но эта мысль справедлива в отношении всякого более-менее существенного программного продукта. Часто бывает так, что программа уже написана и используется некоторое время, и только после этого к ее автору наконец приходит понимание, как же она должна быть устроена.

Настроившись на этот оптимистический лад, приступим к созданию библиотеки для работы с CSV-данными, которую потом придется выбросить, — ее *прототипа*. В первой версии мы будем игнорировать большинство трудных мест, учитываемых в законченном профессиональном продукте, но тем не менее библиотека будет достаточно полной и работоспособной, чтобы на ее примере ознакомиться с проблемой.

Начнем с функции `csvgetline`, которая считывает одну строку CSV-данных из файла в буфер, разбивает ее на поля, помещает поля в массив, удаляет кавычки и возвращает количество полей. В течение многих лет мы программировали нечто подобное практически на всех языках, так что это не новая задача. Ниже приведена пробная версия на С; она помечена вопросительными знаками, потому что это всего лишь прототип, требующий серьезной доработки.

```
?    char buf[200];       /* буфер для строки */
?    char *field[20];     /* поля */
?
?    /* csvgetline: ввод и разбиение строки; возвращает количество
полей */
?    /* пробные данные: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
?    int csvgetline(FILE *fin)
?    {
?        int nfield;
?        char *p, *q;
?
?        if (fgets(buf, sizeof(buf), fin) == NULL)
?            return -1;
?        nfield = 0;
?        for (q = buf; (p=strtok(q, ",\n\r")) != NULL; q = NULL)
?            field[nfield++] = unquote(p);
?        return nfield;
?    }
```

Комментарий перед началом функции содержит образец входных данных для программы; такие комментарии бывают полезны, если программа должна воспринимать и анализировать данные сложного формата.

CSV-данные устроены слишком сложно, чтобы с ними справилась функция `scanf`. Вот почему мы воспользовались библиотечной функцией `strtok`. При каждом вызове `strtok(p,s)` возвращается указатель на первую лексему в строке `p`, не содержащую символов из `s`; функция `strtok` завершает лексему, заменяя следующий символ исходной строки нулевым байтом. При первом вызове первый аргумент `strtok` равен исходной анализируемой строке. При последующих вызовах этот аргумент равен `NULL`, что требует от функции продолжать разбор строки с того

места, где он закончился в прошлый раз. Это довольно посредственный способ. Поскольку неизвестно, где хранится строка между вызовами функции `strtok`, одновременно может быть активна только одна последовательность вызовов. Перемежающиеся асинхронные вызовы могут помешать друг другу.

Функция `unquote` удаляет открывающие и закрывающие кавычки, фигурирующие в приведенных выше пробных данных. Но вложенные кавычки она обрабатывать не умеет, поэтому как прототип она хороша, а до окончательного варианта еще не дотягивает.

```
?    /* unquote: удаление открывающих/закрывающих кавычек */
?    char *unquote(char *p)
?    {
?        if (p[0] == '"') {
?            if (p[strlen(p)-1] == '"')
?                p[strlen(p)-1] = '\0';
?            p++;
?        }
?        return p;
?    }
```

Проверить работу функции `csvgetline` можно с помощью простейшей тестовой программы:

```
?    /* csvtest main: тестирует функцию csvgetline */
?    int main(void)
?    {
?        int i, nf;
?
?        while ((nf = csvgetline(stdin)) != -1)
?            for (i = 0; i < nf; i++)
?                printf("field[%d] = `%s'\n", i, field[i]);
?        return 0;
?    }
```

Функция `printf` заключает поля в одинарные кавычки, что позволяет четко выделить их, а также помогает выявить ошибки при обработке символов пустого пространства.

Запустим эту программу и зададим ей исходные данные, загруженные с Web-сайта программой `getquotes.tcl`:

```
% getquotes.tcl | csvtest
...
field[0] = 'LU'
field[1] = '86.375'
field[2] = '11/5/1998'
field[3] = '1:01PM'
field[4] = '-0.125'
field[5] = '86'
field[6] = '86.375'
field[7] = '85.0625'
field[8] = '2888600'
field[0] = 'T'
field[1] = '61.025'
...
```

(В этом примере отброшены заголовки HTTP.)

Итак, мы имеем прототип, который, похоже, вполне способен работать с данными того типа, который показан выше. Но разумно было бы попробовать его и на других данных тоже, особенно если мы собираемся передавать разработанные функции другим пользователям. Мы нашли еще один Web-сайт с биржевыми котировками и получили файл аналогичной информации, но в несколько другой форме: записи отделяются символами возврата каретки (`\r`) вместо символов конца строки, а в конце файла такие символы не ставятся. Вот эти данные, отформатированные под печатную страницу:

```
"Ticker","Price","Change","Open","Prev Close","Day High",
    "Day Low","52 Week High","52 Week Low","Dividend",
    "Yield","Volume","Average Volume","P/E"
"LU",86.313,-0.188,86.000,86.500,86.438,85.063,108.50,
    36.18,0.16,0.1,2946700,9675000,N/A
"T",61.125,0.938,60.375,60.188,61.125,60.000,68.50,
    46.50,1.32,2.1,3061000,4777000,17.0
"MSFT",107.000,1.500,105.313,105.500,107.188,105.250,
    119.62,59.00,N/A,N/A,7977300,16965000,51.0
```

Получив на вход эти данные, наша пробная программа с позором провалилась. Дело в том, что мы разработали наш прототип на основе одного источника данных и тестировали программу только на этом же источнике. Поэтому не стоит удивляться, что первая же встреча с другими данными закончилась столь плачевно. Длинные строки, множество полей, неожиданные или пропущенные разделители — все это потенциальные источники проблем. Наш черновой прототип может годиться для личного пользования или для демонстрации общего подхода, но не более того. Перед тем как переделывать программу, следует более тщательно подойти к ее проектированию.

При создании прототипа мы уже приняли целый ряд решений, как явных, так и неявных. Ниже перечислены некоторые из сделанных предположений и принятых решений, причем не всегда наилучших для библиотеки общего пользования. Каждое из них порождает проблемы, требующие более пристального внимания.

- Программа не обрабатывает длинные строки данных или большое количество полей. Она дает неправильные ответы или вообще завершается аварийно, потому что даже не проверяет переполнение, не говоря уже о том, чтобы возвращать специальные коды в случае ошибок.
- Предполагается, что исходные данные состоят из строк, разделенных символами конца строки.
- Поля отделяются друг от друга запятыми, а окружающие их кавычки удаляются. Вложенные запятые или кавычки никак не обрабатываются.
- Исходная строка не сохраняется, а затирается в процессе ее разбиения на поля.
- При переходе от одной строки данных к другой не запоминаются никакие данные; если все же что-то нужно будет запомнить, придется сделать копию.
- Обращение к полям выполняется через глобальную переменную (массив `field`), совместно используемую функцией `csvgetline` и функциями, ее вызывающими. Не осуществляется никакого контроля над обращениями к

содержимому полей или указателям на них. Обращение к полям за пределами последнего также никак не блокируется.

- Наличие глобальных переменных делает данную реализацию неподходящей для многопоточных сред или даже для двух последовательностей поочередных вызовов.
- Вызывающий модуль должен сам открывать и закрывать файлы — функция `csvgetline` считывает информацию только из открытых файлов.
- Ввод данных и их разбиение на поля неразрывно связаны: при каждом вызове функции строка автоматически вводится и разбивается на поля независимо от того, необходима ли программе эта операция.
- Возвращаемое значение равно количеству полей в строке; чтобы вычислить это значение, необходимо разбить строку на поля. Также нет способа отличить ошибку от конца файла.
- Не существует другого способа изменить какую-либо из перечисленных характеристик программы, кроме непосредственной модификации исходного кода.

Этот длинный, хотя и неполный список иллюстрирует некоторые из компромиссов, на которые идет программист. Каждое из проектных решений явно или неявно вплетается в код. Для одноразовой задачи — например, считывание данных одного фиксированного формата из одного известного источника — это вполне нормально. Но что если формат изменится, или внутри кавычек появится запятая, или сервер сгенерирует длинную строку с большим количеством полей?

Все эти проблемы кажутся легко решаемыми, поскольку “библиотека” очень маленькая и в любом случае представляет собой всего лишь прототип. Однако представьте себе, что она проваляется на полке несколько месяцев или лет, а затем войдет в состав большой программы, спецификации и интерфейсы которой изменяются со временем. Как тогда приспособится к этой ситуации функция `csvgetline`? Если программа должна использоваться другими людьми, то компромиссные решения, принятые при проектировании прототипа, могут обернуться в отдаленном будущем негативными последствиями. По этому сценарию уже развивалась история многих плохо спроектированных интерфейсов. Следует с прискорбием отметить, что многие быстро и плохо написанные модули кода со временем вошли в популярные программные продукты, оставшись в их составе такими же низкокачественными, но уже гораздо менее быстрыми.

4.3. Библиотека для общего пользования

Вооружившись знаниями о поведении прототипа, приступим к разработке библиотеки, достойной передачи в другие руки. Наиболее очевидное требование — это сделать функцию `csvgetline` более устойчивой к длинным строкам и большому количеству полей. Следует также тщательнее отнестись к разбиению строк данных на поля.

Чтобы разработать интерфейс, которым могли бы пользоваться другие люди, необходимо учесть вопросы, поставленные в начале главы: внешний интерфейс, сокрытие информации, управление ресурсами и обработка ошибок. Соотношение между

этими требованиями сильно влияет на структуру модулей библиотеки. Наша классификация данных требований несколько условна, поскольку все они взаимосвязаны.

Интерфейс. Мы выбрали для реализации три основные операции:

`char *csvgetline(FILE *)`: считывание новой строки CSV-данных;

`char *csvfield(int n)`: получение n-го поля текущей строки;

`int csvnfield(void)`: получение количества полей в текущей строке.

Какое значение должна возвращать функция `csvgetline`? Желательно, чтобы она возвращала как можно больше полезной информации в удобной форме, например, количество полей, как и в прототипе. Но в таком случае количество полей придется вычислять даже в том случае, если сами поля не используются. Еще одно возможное возвращаемое значение — это длина строки исходных данных, на которую влияет то обстоятельство, сохраняется в строке завершающий ее нулевой символ или нет. После ряда экспериментов мы решили, что функция `csvgetline` должна возвращать указатель на исходную строку или `NULL`, если встретился конец файла.

Символ конца строки будет удаляться из строки исходных данных, возвращаемой функцией `csvgetline`, поскольку по необходимости его легко восстановить.

Определить, что такое поле — непростое дело. Мы попытались обобщить наши эмпирические наблюдения, почерпнутые из электронных таблиц и других аналогичных программ. Поле — это последовательность из нескольких символов (в частном случае — ни одного). Поля отделяются друг от друга запятыми. Пробелы в начале и в конце поля сохраняются. Поле может быть заключено в двойные кавычки, и в этом случае в его состав могут входить запятые. Поле в кавычках может также содержать символы двойных кавычек, для обозначения которых используются сразу два таких символа подряд; так, поле `"x""y"` обозначает строку данных `x"y`. Поля могут быть пустыми; например, кавычки без других символов между ними (`""`) как раз обозначают пустое поле, так же как и отсутствие символов между двумя запятыми.

Поля нумеруются начиная с нуля. А что если пользователь запросит несуществующее поле, сделав вызов `csvfield(-1)` или `csvfield(100000)`? Можно возвратить строку `""` (пустое поле), поскольку ее можно выводить на экран или сравнивать с другими строками. Программы, работающие с переменным количеством полей, не обязаны принимать специальные меры по обращению с несуществующими строками. Второй вариант — это вывести сообщение об ошибке или даже завершить программу аварийно. Вскоре мы обсудим, почему это нежелательно. Мы решили возвращать значение `NULL` — обычное обозначение для несуществующей строки в языке C.

Сокрытие информации. Библиотека не должна накладывать никаких ограничений на длину входной строки или количество полей в ней. Чтобы этого добиться, нужно либо заставить вызывающий модуль обеспечить вызываемые функции достаточным количеством памяти, либо вызываемые (библиотечные) модули должны позаботиться о ее распределении. Например, при вызове библиотечной функции `fgets` в нее передается массив и его максимальная длина. Если вводимая строка длиннее буфера, она разбивается на части. Этот способ неприемлем для CSV-интерфейса, поэтому наша библиотека будет сама распределять дополнительное количество памяти, если ее окажется недостаточно.

Таким образом, только функции `csvgetline` известно о выполняемом ею распределении памяти; никакая информация о том, как именно она это делает, не просачивается наружу. Такое сокрытие данных лучше всего осуществить через интерфейс библиотеки: `csvgetline` будет считывать следующую строку, какова бы ни была ее длина, `csvfield(n)` — возвращать указатель на байты `n`-го поля текущей строки, а `csvnfield` — возвращать количество полей в текущей строке.

По мере поступления более длинных строк или большего количества полей придется наращивать объем используемой памяти. Подробности этих операций скрыты в `csv`-функциях, и никакие другие части программы не имеют о них понятия — например, используются ли изначально малые, но растущие массивы, или же массивы очень большой длины, или вообще совершенно другие структуры данных. Интерфейс также скрывает от пользователя момент освобождения памяти.

Если пользователь вызывает только функцию `csvgetline`, то нет необходимости разбивать ее на поля; это можно сделать по специальному требованию. От пользователя скрывается и такая подробность реализации, как разбиение строки на поля: выполняется ли оно немедленно (сразу после считывания строки), по общему требованию (когда запрашивается количество полей или какое-нибудь из них) или по специальному требованию (тогда извлекается только запрашиваемое поле).

Управление ресурсами. Необходимо решить, кто будет отвечать за совместно используемые данные. Должна ли функция `csvgetline` возвращать исходный экземпляр данных или делать их копию? Мы решили, что `csvgetline` будет возвращать указатель на исходную строку данных, которая затирается при следующем считывании. Поля формируются в копии исходной строки, а функция `csvfield` возвращает указатель на конкретное поле в копии строки. В этом случае пользователь должен сам сделать копию, если он хочет сохранить или изменить какое-либо поле или строку. Ответственность за освобождение памяти, когда она становится ненужной, возлагается на пользователя.

Кто должен открывать и закрывать файл исходных данных? Тот, кто его открывает, должен и закрывать: парные задачи должны выполняться на одном и том же уровне или в одном модуле. Предполагается, что функция `csvgetline` будет вызываться с указателем типа `FILE` на уже открытый файл, который вызывающая функция закроет сама в нужный момент.

Управление ресурсами, которые совместно используются или передаются через границу, разделяющую вызывающий модуль и библиотеку, представляет собой сложную задачу. Часто выбор тех или иных предпочтений диктуется хоть и разумными, но весьма противоречивыми соображениями. Ошибки и недоразумения при управлении совместными ресурсами порождают много сбоев в программах.

Обработка ошибок. Поскольку функция `csvgetline` в случае неудачи всегда возвращает значение `NULL`, нет надежного способа отличить конец файла от ошибки — например, нехватки памяти. Аналогично, при обращении к несуществующему полю не возникает состояние ошибки. По аналогии с `ferror` можно было бы добавить к интерфейсу еще одну функцию `csvgeterror` для получения информации о последней сделанной ошибке, но в этой версии библиотеки мы опустили данное средство для простоты.

В принципе, библиотечные функции не должны просто завершаться аварийно, если случается ошибка; тип ошибки необходимо сообщить вызывающему модулю, чтобы тот принял надлежащие меры. Не следует также выдавать сообщения об ошибках или открывать диалоговые окна, поскольку программа может выполняться в среде, где это неуместно и будет мешать работе. Обработка ошибок — это важная тема, которую стоит обсудить отдельно. Мы рассмотрим ее позже в этой же главе.

Спецификация. Принятые ранее решения следует собрать в одном месте в виде спецификации задач, которые должна выполнять функция `csvgetline`, и способов ее использования. В большом проекте спецификация (иначе называемая *техническим заданием*) обязательно предшествует реализации, потому что авторы технического задания и авторы программы — обычно разные люди, часто даже из разных организаций. Тем не менее на практике часто случается, что оба эти вида работ выполняются параллельно: спецификация и код эволюционируют одновременно. Иногда такое “техническое задание” даже пишется после того, как программа уже разработана, фиксируя в виде требований то, что в коде делается на самом деле.

Наилучший подход к делу состоит в том, чтобы написать спецификацию до разработки, а затем вносить в нее изменения по мере накопления опыта реализации. Чем точнее и аккуратнее спецификация, тем больше вероятность, что в результате получится работоспособная программа. Даже для программ личного пользования бывает полезно подготовить достаточно полное техническое задание, поскольку это стимулирует рассматривать возможные альтернативы и фиксировать принятые решения.

Для наших целей нужно иметь спецификацию, содержащую прототипы функций и подробные предписания относительно их поведения, обязанностей и сделанных предположений:

поля должны отделяться запятыми;
поле может заключаться в двойные кавычки ("...");
поле в кавычках может содержать запятые, но не символы конца строки;
поле в кавычках может содержать двойные кавычки, представляемые удвоенными символами ("");
поля могут быть пустыми; пустое поле представляется кавычками ("") или пустой строкой;
в строках сохраняются пробелы до и после фактических данных;

```
char csvgetline(FILE *f);
```

считывает одну строку из открытого входного файла `f`; предполагается, что строки исходных данных оканчиваются на `\r`, `\n`, `\r\n` или `EOF`;

возвращает указатель на строку с удаленными завершающими символами, или `NULL`, если встретился `EOF`;

строки могут иметь произвольную длину; если превышен лимит памяти, возвращается `NULL`;

строка должна восприниматься как объект “только для чтения”; вызывающий модуль должен сделать ее копию, если необходимо внести изменения или сохранить содержимое в другом месте;

`char *csvfield(int n);`

поля нумеруются начиная с `0`;
возвращает `n`-е поле из последней строки, введенной функцией `csvgetline`;
возвращает `NULL`, если `n` < 0 или выходит за пределы последнего поля;
поля отделяются друг от друга запятыми;
поля могут быть окружены кавычками (`"..."`); они удаляются; внутри кавычек двойные их символы (`""`) заменяются одинарными (`"`), а запятая не считается разделителем;
в полях, не взятых в кавычки, кавычки считаются обычными символами;
в строке может быть любое количество полей произвольной длины; если превышен лимит памяти, возвращается `NULL`;
поле должно считаться объектом “только для чтения”; вызывающий модуль должен сделать его копию, если необходимо внести изменения или сохранить содержимое в другом месте;
если функция вызывается до вызова `csvgetline`, результат не определен;

`int csvnfield(void);`

возвращает количество полей в последней строке, считанной функцией `csvgetline`;
если функция вызывается до вызова `csvgetline`, результат не определен.

Эта спецификация все еще оставляет открытыми ряд вопросов. Например, какие значения должны возвращать функции `csvfield` и `csvnfield`, если они вызываются после того, как функция `csvgetline` встретила символ конца файла (`EOF`)? Как следует обрабатывать неправильно сформированные поля данных? Решить с ходу все подобные вопросы не так-то просто даже для небольшой программной системы, а для большой — практически невозможно, хотя необходимо всеми силами пытаться это сделать. Все промахи и недочеты удастся исправить только в ходе реализации программы.

Далее в этом разделе приводится новая реализация функции `csvgetline` и всей библиотеки. Библиотека разбита на два файла: заголовочный файл `csv.h`, содержащий объявления функций и представляющий открытый интерфейс библиотеки, а также файл реализации `csv.c`, содержащий собственно ее код. Пользователи подключают файл `csv.h` к своим программам, а скомпилированный файл `csv.c` компонуется с программой на этапе редактирования внешних связей. Исходный код библиотеки пользователю знать незачем.

Ниже приведен текст заголовочного файла.

```
/* csv.h: интерфейс библиотеки csv */

extern char *csvgetline(FILE *f); /* считывает следующую строку */
extern char *csvfield(int n);     /* возвращает поле n */
extern int csvnfield(void);       /* возвращает количество полей */
```

Внутренние переменные, в которых хранится текст, и внутренние функции наподобие `split` объявлены статическими (с модификатором `static`), так что они ви-

димы только в том файле, который непосредственно их содержит. Это самый простой способ скрыть информацию в программном модуле на языке C.

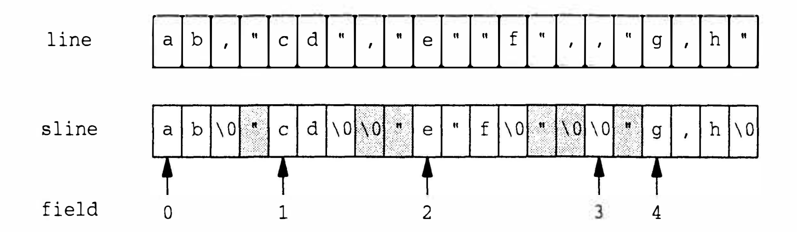
```
enum { NOMEM = -2 };           /* сигнал нехватки памяти */

static char *line    = NULL;  /* вводимая строка */
static char *sline   = NULL;  /* копия строки для split */
static int  maxline  = 0;     /* длина line[] и sline[] */
static char **field  = NULL;  /* указатели на поля */
static int  maxfield = 0;     /* длина field[] */
static int  nfield   = 0;     /* количество полей в field[] */

static char fieldsep[] = ","; /* символы-разделители полей */
```

Переменные также инициализируются статически. Их инициализированные значения используются для проверки того, необходимо ли создавать или расширять массивы.

Данные объявления создают весьма простую структуру данных. Массив `line` содержит вводимую строку; массив `sline` создается путем копирования символов из `line` и вставки маркеров конца строки после каждого поля. Массив `field` содержит указатели на отдельные поля в массиве `sline`. Ниже на схеме показаны эти три массива после ввода и обработки строки `ab,"cd","e""f",,"g,h"`. Затененные элементы в массиве `sline` не входят в состав полей.



А вот и сама функция `csvgetline`:

```
/* csvgetline:  ввод строки, по необходимости расширение */
/* пробные данные: "LU",86.25,"11/4/1998","2:19PM",+4.0625 */
char *csvgetline(FILE *fin)
{
    int i, c;
    char *newl, *news;

    if (line == NULL) {    /* выделить память при 1-м вызове */
        maxline = maxfield = 1;
        line = (char *) malloc(maxline);
        sline = (char *) malloc(maxline);
        field = (char **) malloc(maxfield*sizeof(field[0]));
        if (line == NULL || sline == NULL || field == NULL) {
            reset();
            return NULL;  /* не хватает памяти */
        }
    }
    for (i=0; (c=getc(fin))!=EOF && !endofline(fin,c); i++) {
        if (i >= maxline-1) {  /* расширение строки */
```

```
                maxline *= 2;       /* удвоение размера */
                newl = (char *) realloc(line, maxline);
                news = (char *) realloc(sline, maxline);
                if (newl == NULL || news == NULL) {
                    reset();
                    return NULL;    /* не хватает памяти */
                }
                line = newl;
                sline = news;
            }
            line[i] = c;
        }
        line[i] = '\0';
        if (split() == NOMEM) {
            reset();
            return NULL;            /* не хватает памяти */
        }
        return (c == EOF && i == 0) ? NULL : line;
    }
```

Введенные данные накапливаются в строке `line`, которая расширяется по мере необходимости путем вызова функции `realloc`. При каждом расширении длина этой строки увеличивается вдвое, как в разделе 2.6. Размер массива `sline` поддерживается таким же, как и `line`; функция `csvgetline` вызывает функцию `split` для создания указателей на поля и помещения их в отдельный массив `field`, который также расширяется по мере необходимости.

По нашему обыкновению мы начинаем работу с очень коротких массивов и расширяем их по требованию. Тем самым гарантируется, что код для расширения массивов обязательно будет выполняться. Если распределение памяти терпит неудачу, вызывается функция `reset`, которая восстанавливает глобальные переменные в их первоначальном состоянии, так что последующий вызов `csvgetline` имеет шанс пройти успешно:

```
/* reset: восстановление значений переменных */
static void reset(void)
{
    free(line);  /* free(NULL) разрешено в ANSI C */
    free(sline);
    free(field);
    line = NULL;
    sline = NULL;
    field = NULL;
    maxline = maxfield = nfield = 0;
}
```

Функция `endofline` обрабатывает случаи, когда входная строка заканчивается возвратом каретки, символом конца строки, обоими этими символами, или даже EOF:

```
/* endofline: обработка \r, \n, \r\n или EOF */
static int endofline(FILE *fin, int c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
```

```
        if (c == '\r') {
            c = getc(fin);
            if (c != '\n' && c != EOF)
            ungetc(c, fin);  /* лишнее; возвращаем c */
        }
        return eol;
    }
```

Отдельная функция подобного рода необходима по той причине, что стандартные функции ввода не способны обработать огромное разнообразие форматов, встречающихся на практике.

В нашем прототипе для получения следующей лексемы использовалась функция `strtok`, которая искала в потоке символ-разделитель наподобие запятой. Но ее использование стало невозможным из-за потенциального наличия запятых внутри кавычек. Поэтому необходимо серьезно изменить реализацию функции `split`, хотя ее интерфейс менять не обязательно. Рассмотрим следующие строки исходных данных:

```
    "",,""
    ,"",
    ,,
```

Каждая из этих строк содержит три пустых поля. Чтобы гарантировать корректную обработку таких строк и других подобных данных функцией `split`, необходимо значительно усложнить эту функцию. Это тот случай, когда обработка именно особых случаев и предельных вариантов начинает доминировать в общем объеме программы.

```
    /* split: разбивает строку на поля */
    static int split(void)
    {
        char *p, **newf;
        char *sepp; /* указатель на временный разделитель */
        int sepc;   /* временный символ-разделитель */

        nfield = 0;
        if (line[0] == '\0')
            return 0;
        strcpy(sline, line);
        p = sline;

        do {
            if (nfield >= maxfield) {
                maxfield *= 2;          /* удвоение длины */
                newf = (char **) realloc(field,
                            maxfield * sizeof(field[0]));
                if (newf == NULL)
                    return NOMEM;
                field = newf;
            }
            if (*p == '"')
                sepp = advquoted(++p);  /* пропуск кавычки */
            else
                sepp = p + strcspn(p, fieldsep);
            sepc = sepp[0];
            sepp[0] = '\0';             /* символ конца */
```

```
            field[nfield++] = p;
            p = sepp + 1;
        } while (sepc == ',');

        return nfield;
    }
```

Вот что происходит в цикле. По мере необходимости наращивается массив указателей на поля, затем для нахождения и обработки следующего поля вызывается одна из двух других функций. Если поле начинается с кавычки, то функция `advquoted` распознает поле и возвращает указатель на разделитель, который завершает поле. В противном случае ищется следующая запятая; для этого используется библиотечная функция `strcspn(p,s)`, которая находит в строке `p` первое вхождение любого символа из строки `s`. Эта функция возвращает количество пропущенных символов.

Кавычки внутри поля представляются удвоенными символами кавычек, поэтому функция `advquoted` укорачивает такие символы до одного. Она также удаляет кавычки, в которые заключено поле. Некоторую сложность функции придает попытка корректно справиться с некорректными входными данными, такими как `"abc"def`. В подобных случаях мы включаем в состав поля все символы, следующие за второй кавычкой, пока не встретится корректный символ-разделитель. В Microsoft Excel, по-видимому, используется тот же алгоритм.

```
/* advquoted: поле в кавычках; возвращается указатель
              на следующий разделитель */
static char *advquoted(char *p)
{
    int i, j;

    for (i = j = 0; p[j] != '\0'; i++, j++) {
        if (p[j] == '"' && p[++j] != '"') {
            /* копируем до следующего разделителя или \0 */
            int k = strcspn(p+j, fieldsep);
            memmove(p+i, p+j, k);
            i += k;
            j += k;
            break;
        }
        p[i] = p[j];
    }
    p[i] = '\0';
    return p + j;
}
```

Функции `csvfield` и `csvnfield` тривиальны, поскольку входная строка уже разбита на поля к моменту их вызова:

```
/* csvfield: возвращает указатель на n-е поле */
char *csvfield(int n)
{
    if (n < 0 || n >= nfield)
        return NULL;
    return field[n];
}
```

```
/* csvnfield:  возвращает количество полей */
int csvnfield(void)
{
    return nfield;
}
```

Наконец, модифицируем тестирующую программу так, чтобы она работала с новой версией библиотеки. Поскольку в новой версии хранится копия введенной строки (а в прототипе этого не было), перед выводом полей можно вывести и саму исходную строку:

```
/* csvtest main: тестирует библиотеку CSV */
int main(void)
{
    int i;
    char *line;

    while ((line = csvgetline(stdin)) != NULL) {
        printf("line = `%s'\n", line);
        for (i = 0; i < csvnfield(); i++)
            printf("field[%d] = `%s'\n", i, csvfield(i));
    }
    return 0;
}
```

На этом наша С-версия библиотеки окончена. Она может обрабатывать входные данные произвольной длины и выполняет осмысленные операции даже над некорректными данными. За это пришлось заплатить вчетверо большей длиной, чем у первоначального прототипа, и повышенной сложностью некоторых участков кода. Увеличение размеров и сложности — это обычное явление при переходе от пробной версии программы к профессиональной.

Упражнение 4.1. Имеется несколько возможных уровней сложности при реализации разбиения полей: можно выделить все поля сразу, но по запросу к конкретному полю, или же только запрошенное поле, или все поля вплоть до запрошенного. Составьте список всех этих возможностей, оцените их преимущества и степень сложности, затем реализуйте и сравните быстродействие.

Упражнение 4.2. Добавьте новые функциональные возможности для того, чтобы разделители можно было заменить (а) на произвольный класс символов; (б) на различные разделители для различных полей; (в) на регулярные выражения (подробно об этом речь пойдет в главе 9). Как должен выглядеть интерфейс в этих случаях?

Упражнение 4.3. В нашем варианте используется статическая инициализация в зависимости от ключевого значения: если указатель на позицию равен `NULL`, то выполняется инициализация. Другой вариант — это потребовать от пользователя вызывать явную функцию инициализации, в которой можно было бы задавать начальные длины массивов. Реализуйте версию библиотеки, сочетающую преимущества обоих подходов. Какова роль функции `reset` в этой реализации?

Упражнение 4.4. Спроектируйте и реализуйте библиотеку для создания данных в CSV-формате. Простейшая ее версия может принимать массив строк и выводить их с кавычками и запятыми. В более совершенной версии могла бы использоваться строка формата, аналогичная `printf`. Некоторые предложения по системе обозначений можно найти в главе 9.

4.4. Реализация на языке C++

В этом разделе будет разработана версия библиотеки на языке C++, в которой будут учтены некоторые ограничения, присущие C-версии. Вначале мы внесем изменения в спецификацию библиотеки, самое важное из которых состоит в замене символьных массивов C на строки C++. Использование строкового типа C++ автоматически решит некоторые проблемы управления памятью, поскольку библиотечные функции возьмут эту задачу на себя. В частности, функции работы с полями будут возвращать строки, доступные для модификации вызывающим модулем. Это более гибкая реализация алгоритма, чем предыдущая.

В классе `Csv` будет определен открытый интерфейс и аккуратно скрыты все переменные и функции реализации. Поскольку объект класса содержит все параметры своего состояния, можно объявлять несколько переменных-экземпляров класса `Csv`, независимых друг от друга, и работать с несколькими потоками CSV-данных сразу.

```
class Csv {  // считывание данных, разделенных запятыми
    // пробные данные: "LU",86.25,"11/4/1998","2:19PM",+4.0625

  public:
    Csv(istream& fin = cin, string sep = ",") :
        fin(fin), fieldsep(sep) {}

    int getline(string&);
    string getfield(int n);
    int getnfield() const { return nfield; }

  private:
    istream& fin;           // указатель на входной файл
    string line;            // входная строка
    vector<string> field;   // строки-поля
    int nfield;             // количество полей
    string fieldsep;        // символы-разделители

    int split();
    int endofline(char);
    int advplain(const string& line, string& fld, int);
    int advquoted(const string& line, string& fld, int);
};
```

Параметры, заданные по умолчанию для конструктора определены так, чтобы объект `Csv` считывал данные из стандартного потока ввода и использовал обычные разделители полей. Вместо этих параметров можно явно указать другие.

Для работы со строками используются стандартные классы C++ string и vector, а не обычные строки в стиле С. У объекта типа string нет состояния “не существует” — если объект пуст, его длина всего-навсего равна нулю. Аналога значения NULL также не существует, поэтому его нельзя использовать в качестве маркера конца файла. В итоге метод Csv::getline возвращает введенную строку исходных данных через аргумент по адресу, а возвращаемое значение резервируется для сообщений об ошибках и конце файла.

```
// getline: вводит строку, по необходимости расширяя ее
int Csv::getline(string& str)
{
    char c;

    for (line = ""; fin.get(c) && !endofline(c); )
        line += c;
    split();
    str = line;
    return !fin.eof();
}
```

Знак операции += перегружен таким образом, чтобы добавлять к строке символ.

Функция endofline требует некоторой модификации. И снова придется считывать символы по одному, поскольку ни одна стандартная функция не может учесть все особенности наших входных данных.

```
// endofline: обрабатывает \r, \n, \r\n или EOF
int Csv::endofline(char c)
{
    int eol;

    eol = (c=='\r' || c=='\n');
    if (c == '\r') {
        fin.get(c);
        if (!fin.eof() && c != '\n')
            fin.putback(c);  // возврат назад
    }
    return eol;
}
```

А это новая версия функции split:

```
// split: разбивает строку на поля
int Csv::split()
{
    string fld;
    int i, j;

    nfield = 0;
    if (line.length() == 0)
        return 0;
    i = 0;

    do {
        if (i < line.length() && line[i] == '"')
            j = advquoted(line, fld, ++i); // пропуск кавычки
```

```
            else
                j = advplain(line, fld, i);
            if (nfield >= field.size())
                field.push_back(fld);
            else
                field[nfield] = fld;
            nfield++;
            i = j + 1;
        } while (j < line.length());

        return nfield;
    }
```

Поскольку функция strcspn не работает со строками C++, необходимо внести изменения как в split, так и в advquoted. Новая версия функции advquoted использует стандартную функцию C++ find_first_of для поиска следующего вхождения символа-разделителя. При ее вызове в виде s.find_first_of(fieldsep,j) в строке s разыскивается первое вхождение любого символа из строки fieldsep, начиная с позиции j включительно. Если такое вхождение найти не удается, возвращается индекс за пределами конца строки, поэтому его нужно вернуть назад в допустимый диапазон. Во внутреннем цикле for, следующем за этой операцией, символы поля накапливаются с помощью переменной fld вплоть до достижения символа-разделителя.

```
// advquoted: обрабатывает поле в кавычках;
//   возвращает индекс следующего разделителя
int Csv::advquoted(const string& s, string& fld, int i)
{
    int j;

    fld = "";
    for (j = i; j < s.length(); j++) {
        if (s[j] == '"' && s[++j] != '"') {
            int k = s.find_first_of(fieldsep, j);
            if (k > s.length())  // разделитель не найден
                k = s.length();
            for (k -= j; k-- > 0; )
                fld += s[j++];
            break;
        }
        fld += s[j];
    }
    return j;
}
```

Функция find_first_of также используется в новой функции advplain, которая сдвигает указатель позиции через следующее поле, не заключенное в кавычки. Это изменение так же необходимо, как и предыдущие, потому что строковые функции C наподобие strcspn неприменимы к строкам C++, представляющим собой совершенно иной тип данных.

```
// advplain: обрабатывает поле без кавычек;
//   возвращает индекс следующего разделителя
int Csv::advplain(const string& s, string& fld, int i)
```

```
{
    int j;

    j = s.find_first_of(fieldsep, i); // поиск разделителя
    if (j > s.length())               // не найден
        j = s.length();
    fld = string(s, i, j-i);
    return j;
}
```

Как и ранее, функция `Csv::getfield` тривиальна, а `Csv::getnfield` вообще настолько проста, что определена прямо в объявлении класса.

```
// getfield: возвращает n-е поле
string Csv::getfield(int n)
{
    if (n < 0 || n >= nfield)
        return "";
    else
        return field[n];
}
```

Наша тестовая программа представляет собой простейшую вариацию предыдущей:

```
// Csvtest main: тестирует класс Csv
int main(void)
{
    string line;
    Csv csv;

    while (csv.getline(line) != 0) {
        cout << "line = `" << line <<"'\n";
        for (int i = 0; i < csv.getnfield(); i++)
            cout << "field[" << i << "] = `"
                 << csv.getfield(i) << "'\n";
    }
    return 0;
}
```

Способ использования этой библиотеки несколько отличается от версии на языке C, хотя и незначительно. В зависимости от компилятора версия на C++ может работать медленнее как на 40%, так и в четыре раза, если входные данные имеют большой объем наподобие 30 000 строк с 25 полями на строку. Как мы уже говорили в связи с программой `markov`, эти вариации в быстродействии отражают качество реализации библиотеки. Исходный текст программы на C++ примерно на 20% короче, чем на C.

Упражнение 4.5. Усовершенствуйте реализацию на языке C++, перегрузив знак операции `[]`. Эта операция должна давать возможность обращаться к полям через выражения типа `csv[i]`.

Упражнение 4.6. Напишите библиотеку CSV на языке Java, а затем сравните три реализации по их доступности в понимании, надежности и быстродействию.

Упражнение 4.7. Переоформите версию библиотеки CSV на языке C++ в виде шаблона STL.

Упражнение 4.8. Версия библиотеки на C++ позволяет нескольким независимым экземплярам `Csv` работать параллельно без взаимного вмешательства. Это достигается тем, что все параметры состояния инкапсулируются в одном объекте, и таких объектов можно создавать сколько угодно. Измените C-версию библиотеки так, чтобы это стало возможно и в ней, заменив глобальные структуры данных на динамически размещаемые в памяти и инициализируемые явной функцией `csvnew`.

4.5. Принципы интерфейса

В предыдущих разделах мы занимались проработкой деталей интерфейса — подробно определенной и описанной границы между кодом, обеспечивающим предоставление определенных услуг, и кодом, в котором эти услуги используются. Интерфейс определяет, что тот или иной модуль кода должен делать для пользователя, как его функции и (иногда) элементы данных могут использоваться остальными частями программы. В нашем интерфейсе CSV определены три функции — считывание строки, получение поля и возвращение количества полей. Других операций пользователь выполнить не сможет.

Чтобы иметь успех, интерфейс должен хорошо подходить для той работы, которую он делает, т.е. быть простым, общим, стандартизированным, предсказуемым, надежным в работе. Кроме того, он должен легко адаптироваться к другим пользователям и изменениям внутренней реализации. В хороших интерфейсах соблюдается ряд принципов. Система этих принципов не является полностью согласованной, а сами они не вполне независимы, но с их помощью можно более-менее описать, что и как должно происходить на границе между двумя модулями кода.

Скрывайте подробности реализации. Реализация, стоящая за интерфейсом, должна быть скрыта от остальных частей программы так, чтобы ее изменение не повлияло ни на что за ее пределами. Этот организационный принцип известен под многими именами: сокрытие информации, инкапсуляция, абстрагирование, модуляризация и т.д. Все это разные названия для одной и той же идеи. Интерфейс должен скрывать детали реализации, несущественные для клиента (пользователя). Невидимые подробности можно легко изменить — например, для того, чтобы расширить область действия интерфейса, сделать его более эффективным или вообще заменить всю реализацию полностью.

Стандартные библиотеки большинства языков программирования представляют собой хорошо знакомые примеры этого подхода, хотя они и не всегда идеально спроектированы. Стандартная библиотека ввода-вывода C — одна из самых известных и наилучших. Она состоит из двух-трех десятков функций для открытия, закрытия, чтения, записи и других операций с файлами. Реализация подробностей файлового ввода-вывода скрыта в структурном типе `FILE*`, свойства которого можно посмотреть (он объявлен в файле `<stdio.h>`), но не стоит трогать.

Если в заголовочном файле нет подробного объявления структуры, а только ее имя, то такой тип часто называют *скрытым* (*opaque*), потому что его свойства невидимы, а все операции выполняются через указатель на какой-то реальный объект.

Избегайте глобальных переменных. Везде, где это возможно, передавайте ссылки на все данные через аргументы.

Мы решительно настроены против открытости или общедоступности данных в любой форме — слишком уж сложно поддерживать корректную последовательность обработки данных, если дать пользователю право изменять переменные по его усмотрению. Интерфейсы функций облегчают задачу ограничения доступа, но тем не менее данный принцип часто нарушается. Стандартные потоки ввода-вывода наподобие `stdin` и `stdout` почти всегда определены как элементы глобального массива структур типа `FILE`:

```
extern FILE    __iob[_NFILE];
#define stdin  (&__iob[0])
#define stdout (&__iob[1])
#define stderr (&__iob[2])
```

Эти объявления делают реализацию насквозь видимой. Становится также ясно, что нельзя присваивать какие-либо значения идентификаторам `stdin`, `stdout` и `stderr`, хотя они внешне и похожи на имена переменных. Примечательное имя `__iob` следует соглашению стандарта ANSI C о том, что все закрытые имена реализации, видимые снаружи, должны начинаться с двух знаков подчеркивания. Это снижает вероятность конфликта имен.

Классы в C++ и Java представляют собой гораздо лучшие механизмы сокрытия информации. Понимать их суть очень важно для правильного программирования на этих языках. Классы-контейнеры стандартной библиотеки шаблонов STL в языке C++ заходят в своем следовании принципу еще дальше: кроме некоторых гарантий быстродействия, никакой другой информации о реализации нет вообще, и разработчики библиотеки могут использовать любой механизм по своему выбору.

Выберите небольшой набор непересекающихся примитивов. Интерфейс должен предоставлять столько функциональных возможностей, сколько необходимо, но не более того, и функции не должны пересекаться в выполняемых ими задачах. Имея в библиотеке много функций, ее можно сделать более удобной для использования — все, что пользователь пожелает, будет всегда под рукой. Но большой интерфейс труднее реализовывать и дорабатывать, и из-за его размера может оказаться невозможным сколько-нибудь плодотворно изучить и использовать такой интерфейс. “Интерфейсы разработки приложений”, или API, иногда имеют такие огромные размеры, что нормальному человеку не стоит и надеяться их освоить.

В целях удобства некоторые интерфейсы предлагают несколько способов сделать одно и то же. Это тенденция, которой следует избегать. Стандартная библиотека ввода-вывода C содержит по меньшей мере четыре разные функции для записи одного символа в поток вывода:

```
char c;
putc(c, fp);
fputc(c, fp);
fprintf(fp, "%c", c);
fwrite(&c, sizeof(char), 1, fp);
```

Если поток вывода представляет собой `stdout`, то есть еще несколько возможностей. Все они удобны, но отнюдь не необходимы.

Интерфейсы небольшого объема предпочтительнее, чем большие, — по крайней мере, до тех пор, пока не будет убедительно доказана недостаточность имеющихся функций. Сделайте что-то одно, но сделайте это хорошо. Не добавляйте в интерфейс что-либо только потому, что это можно сделать, и не занимайтесь исправлением интерфейса, если ошибка кроется в реализации. Например, вместо двух функций — быстрой memcpy и безопасной memmove — стоило бы иметь одну, всегда надежную и при этом достаточно быструю во всех ситуациях, когда это возможно.

Не делайте ничего за спиной пользователя. Библиотечная функция не должна записывать секретных файлов или переменных, а также изменять глобальные данные. Кроме того, ей следует быть крайне осторожной с модификацией данных в вызывающем модуле. А вот функция strtok нарушает сразу несколько из этих требований. Неприятно удивляет тот факт, что она записывает нулевые байты в середину строки ее исходных данных. То, что функция использует нулевой байт для обозначения места окончания предыдущей операции, подразумевает хранение неких секретных данных между ее вызовами, а это потенциальный источник ошибок и препятствие к параллельной работе нескольких экземпляров функции. Лучше было бы включить в библиотеку одну функцию для разбиения входной строки на лексемы. По аналогичным причинам вторая С-версия нашей библиотеки не может использоваться с двумя входными потоками (см. упражнение 4.8).

При работе с одним интерфейсом нельзя требовать подключения другого просто потому, что так удобнее разработчику. Вместо этого следует делать интерфейсы самодостаточными, а если это невозможно, то хотя бы информировать пользователя с предельной ясностью, какие еще требуются внешние функциональные возможности. В противном случае бремя технической поддержки и доработки интерфейса ложится на клиента. Очевидный пример — это тяжкая необходимость включать длинные списки заголовочных файлов в код на языках С и С++; заголовочные файлы могут содержать тысячи строк и подключать еще десятки других заголовочных файлов.

Выполняйте одну и ту же операцию везде одинаковым способом. Очень важно соблюдать согласованность и единообразие. Аналогичные задачи должны выполняться аналогичными средствами. Библиотечные функции С из семейства str... очень просты в использовании даже без документации, потому что все они устроены примерно одинаково: данные передаются справа налево, т.е. в том же направлении, что и при присваивании, и все функции возвращают результирующую строку. А вот в стандартной библиотеке ввода-вывода С трудно предсказать порядок аргументов при вызове функций. В некоторых аргумент типа FILE* стоит первым, в некоторых — последним; размер и количество элементов передаются в какой угодно последовательности. В противоположность этому, алгоритмы контейнеров STL обеспечивают весьма единообразный интерфейс, так что становится легко пользоваться даже новой или малознакомой функцией.

Следует добиваться не только внутренней согласованности, но и внешней, т.е. стараться, чтобы функциональные средства использовались по определенному стандарту. Например, функции mem... из библиотеки С были разработаны после функций str..., но заимствовали их стиль. Способ использования стандартных функций ввода-вывода fread и fwrite было бы легче запомнить, если бы он совпадал с использо-

ванием функций read и write, на которых они основаны. Ключи командной строки Unix начинаются со знака "минус", но любой конкретный буквенный ключ может означать совершенно разные вещи даже в близкородственных программах.

Если шаблоны подстановки наподобие * в *.exe раскрываются системным командным интерпретатором, то операции выполняются единообразно. Если же это делается индивидуально прикладными программами, то возможны вариации. В Web-браузерах требуется один щелчок мышью для перехода по гиперссылке, но в других приложениях для активизации программы или перехода по ссылке используется двойной щелчок. В результате многие люди автоматически щелкают дважды.

Перечисленным принципам легче следовать в одной среде и труднее в другой, но все же они сохраняют свое значение везде. Например, скрыть подробности реализации в языке C трудно, но хороший программист постарается их не афишировать, чтобы закрытая информация не стала частью интерфейса и не нарушила принцип сокрытия данных. Комментарии в заголовочных файлах, имена особого формата (например, __iob) и т.д. — все это помогает поощрять хороший стиль программирования там, где не удается заставить пользователя следовать ему.

Несмотря на все это, наши способности по усовершенствованию интерфейсов все же имеют границы. Даже наилучшие интерфейсы сегодняшнего дня могут породить проблемы с наступлением дня завтрашнего. И все же именно хорошее проектирование интерфейсов помогает оттянуть этот зловещий "завтрашний день".

4.6. Управление ресурсами

Одна из наиболее сложных проблем при разработке интерфейса библиотеки (а также класса или пакета) — это управление ресурсами, которыми библиотека владеет либо единолично, либо совместно с вызывающими модулями. Самый очевидный из таких ресурсов — это память, относительно которой всегда возникает вопрос, кто должен ее распределять и освобождать. Но есть и другие важные ресурсы совместного пользования, такие как открытые файлы и общие переменные. Вкратце говоря, связанные с этим проблемы подразделяются на категории проблем инициализации, хранения состояния, совместного обращения и копирования, а также очистки или удаления.

В прототипе нашего пакета CSV используется статическая инициализация указателей, счетчиков и т.п. Но это ограничивает наши возможности, поскольку делает невозможным возврат в первоначальное состояние и запуск библиотеки заново после вызова хотя бы одной функции. Альтернативный подход здесь таков — определить функцию инициализации, устанавливающую все внутренние переменные в корректное исходное состояние. Это позволяет начать операции с начала, но требует от пользователя вызывать функцию инициализации явным образом. Как раз для этой цели функцию reset во второй версии библиотеки можно было бы сделать открытой.

В языках C++ и Java для инициализации полей данных в классах используются конструкторы. Хорошо определенные конструкторы гарантируют, что все данные инициализированы и нет никакого способа создать неинициализированный объект класса. Для поддержки различных видов инициализации объявляется группа конструкторов. Например, в класс Csv можно было бы включить один конструктор, который в качестве аргумента принимает имя файла, а другой — поток ввода.

А как насчет копирования информации, которой владеет и управляет библиотека, например, входных строк и их полей? Наша С-версия программы csvgetline предоставляет прямой доступ к входным строкам (как к самой строке, так и к ее полям), возвращая указатели. Этот неограниченный доступ имеет ряд недостатков. Вполне возможно, что пользователь затрет участок памяти и сделает часть данных непригодными. Возьмем следующий оператор:

```
strcpy(csvfield(1), csvfield(2));
```

Эта конструкция может натворить бед разными способами, например, затереть начало поля №2, если второе поле длиннее первого. Пользователю библиотеки придется делать копию информации, которую он хотел бы сохранить, в промежутке между вызовами csvgetline. В следующем фрагменте кода указатель вполне может стать некорректным, если при втором вызове csvgetline произойдет перераспределение памяти для строкового буфера:

```
char *p;

csvgetline(fin);
p = csvfield(1);
csvgetline(fin);
/* здесь p может стать некорректным */
```

Версия на C++ безопаснее, поскольку копируются такие строки, которые можно изменять как угодно.

В Java для обращения к объектам используются ссылки; объект — это любые данные с общим именем, кроме элементов простых типов наподобие int. Это более эффективно, чем копирование, но кто-нибудь может впасть в заблуждение, что ссылка — это и есть копия. У нас уже была такая ошибка в одной из ранних версий программы markov на Java. (В языке С это вообще вечный источник проблем со строками символов.) Методы-клоны предоставляют возможность делать копии по мере необходимости.

Противоположностью инициализации или конструированию объектов служит их уничтожение, или ликвидация, — очистка данных и возвращение ресурсов в систему после того, как объект больше не нужен. Это особенно важно в отношении памяти, поскольку программа, забывающая о необходимости освобождать память, в конце концов страдает от ее нехватки. Многие современные программы склонны вести себя подобным образом. Аналогичные проблемы возникают с закрытием открытых файлов: если данные буферизуются, то для буфера может требоваться очистка (а для памяти — возвращение в систему). В стандартных библиотечных функциях С очистка буфера происходит автоматически при нормальном завершении программы. Случай аварийного завершения необходимо программировать особо. Стандартная функция С и C++ с именем atexit позволяет получить управление в свои руки непосредственно перед нормальным завершением программы. Разработчики реализаций интерфейсов могут воспользоваться этим средством для выполнения операций по очистке и удалению объектов.

Освобождайте ресурс на том же уровне, на котором он запрашивался. Один из подходов к управлению размещением и освобождением ресурса заключается в том, чтобы его уничтожением ведала та же библиотека, пакет или интерфейс, которые отвечают за распределение памяти для него. Другая формулировка этого подхода такова — интерфейс не должен изменять статус существования ресурса так, чтобы это было заметно снаружи. Наши CSV-библиотеки считывают данные из файлов, которые уже были открыты ранее, и оставляют их открытыми после завершения операций. Пусть модуль, вызывающий библиотеку, сам закрывает свои файлы.

Конструкторы и деструкторы C++ очень помогают следовать этому принципу. При выходе экземпляра класса за пределы его области действия или при его принудительном уничтожении автоматически вызывается деструктор. В деструкторе можно очищать буферы, возвращать память в систему, переустанавливать значения переменных состояния и делать все, что еще может оказаться необходимым. В Java аналогичного механизма нет. Хотя для класса Java можно определить метод ликвидации, нет никакой гарантии, что он вообще будет выполняться, не говоря уже о том, чтобы он выполнялся когда нужно. Поэтому очистку нельзя гарантировать, хотя иногда полезно бывает предположить, что она все-таки выполняется.

И все же в Java существуют удобные средства, сильно облегчающие управление памятью. Это встроенные средства *сборки мусора* (*garbage collection*). По мере выполнения программы она размещает в памяти все новые объекты. Удалить их явным образом невозможно, однако исполнительная система следит за тем, какие объекты используются, а какие — уже нет, и периодически возвращает неиспользуемые объекты в пул свободной памяти.

В деле сборки мусора существует целый ряд подходов и приемов. В некоторых схемах подсчитывается количество обращений к каждому объекту из модулей — ведется его *счетчик ссылок*. Объект освобождается тогда, когда его счетчик ссылок становится равным нулю. Эту технику можно запрограммировать в C и C++ явным образом для управления объектами общего пользования. В других методах периодически отслеживается путь от пула размещенных в памяти объектов ко всем объектам, на которые имеются ссылки. Объекты, для которых этот путь прослежен, считаются используемыми, тогда как объекты, на которые не ссылаются никакие другие, полагаются неиспользуемыми и могут быть освобождены.

Существование автоматической сборки мусора вовсе не означает, что при проектировании и реализации интерфейса не будет никаких проблем с распределением памяти. По-прежнему необходимо определить, должны ли интерфейсы возвращать ссылки на совместно используемые объекты или их копии, и все это существенно влияет на поведение программы. Сборка мусора не обходится даром — необходимо где-то хранить служебную информацию и как-то востребовать назад память; к тому же сборка мусора выполняется в непредсказуемые моменты времени.

Все эти проблемы становятся еще сложнее, если библиотека должна использоваться в среде, где несколько потоков могут одновременно вызывать ее функции, как это делается в многопоточных программах на Java.

Во избежание проблем необходимо писать *реентерабельный* код, т.е. такой, который работает корректно независимо от количества одновременно вызванных модулей. В реентерабельном коде нет ни глобальных переменных, ни статических локальных переменных, и вообще любых переменных, которые могут модифициро-

ваться в то время, пока ими пользуется другой поток. Ключ к хорошему многопоточному программированию лежит в таком разделении компонентов, при котором они взаимодействуют только через четко определенные интерфейсы. Библиотеки, которые неосторожно предоставляют переменные в общее пользование, наносят смертельный удар по этой модели. (В многопоточной программе функция `strtok` — это настоящее стихийное бедствие, как и все функции из библиотеки С, которые хранят данные во внутренних статических переменных.) Если уж так необходимо отдавать переменные в общее пользование, то их необходимо снабдить каким-то механизмом захвата и блокирования, чтобы гарантировать единовременное обращение со стороны только одного потока. Классы оказывают в этом деле существенную помощь, поскольку позволяют реализовать удобные модели совместного использования и блокирования доступа. Синхронизированные методы в Java позволяют одному потоку блокировать целый класс или экземпляр класса от одновременного модифицирования другим классом; механизм синхронизированных блоков позволяет только одному потоку за раз выполнять тот или иной фрагмент кода.

Короче говоря, необходимость многопоточной обработки данных добавляет сложностей программисту, и это слишком большая тема, чтобы сколько-нибудь подробно обсуждать ее здесь.

4.7. Обработка ошибок

В предыдущих главах для обработки ошибок использовались такие функции, как `eprintf` или `estrdup`. В случае ошибки они выводят на экран сообщение и завершают выполнение программы. Например, функция `eprintf` делает все то же самое, что и `fprintf(stderr, ...)`, но в случае ошибки выводит сообщение о ней и завершает программу, возвращая в систему код ошибки. В ее работе используются заголовочный файл `<stdarg.h>` и библиотечная функция `vfprintf`; с их помощью на экран выводятся аргументы, представленные в прототипе многоточием (`...`). Библиотеку `stdarg` необходимо инициализировать вызовом функции `va_start`, а завершить ее работу следует вызовом `va_end`. В главе 9 будут представлены дополнительные примеры работы с этим интерфейсом.

```
#include <stdarg.h>
#include <string.h>
#include <errno.h>

/* eprintf: выводит сообщение об ошибке и выходит */
void eprintf(char *fmt, ...)
{
    va_list args;

    fflush(stdout);
    if(progname() != NULL)
       fprintf(stderr, "%s: ", progname());

    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);
```

```
    if (fmt[0] != '\0' && fmt[strlen(fmt-1)] == ':')
        fprintf(stderr, " %s", strerror(errno));
    fprintf(stderr, "\n");
    exit(2); /* условный код аварийного выхода */
}
```

Если строка формата заканчивается двоеточием, то функция `eprintf` вызывает стандартную функцию C `strerror`, которая возвращает строку с имеющейся в системе дополнительной информацией о возникшей ошибке. Мы также написали функцию `weprintf`, аналогичную `eprintf`, которая отображает предупреждение, но не завершает работу программы. Интерфейс, аналогичный `printf`, удобен для конструирования строк, которые могут посылаться на печать или отображаться в диалоговых окнах.

Аналогичным образом, функция `estrdup` пытается создать копию строки, а затем выходит с выводом сообщения (для чего служит функция `eprintf`), если произошла ошибка распределения памяти:

```
/* estrdup: создает копию строки, сообщает об ошибке */
char *estrdup(char *s)
{
    char *t;

    t = (char *) malloc(strlen(s)+1);
    if (t == NULL)
        eprintf("estrdup(\"%.20s\") failed:", s);
    strcpy(t, s);
    return t;
}
```

Функция `emalloc` делает то же самое при распределении памяти с помощью `malloc`:

```
/* emalloc: вызывает malloc, сообщает об ошибке */
void *emalloc(size_t n);
{
    void *p;

    p = malloc(n);
    if (p == NULL)
        eprintf("malloc of %u bytes failed:", n);
    return p;
}
```

Все эти функции объявляются в одном заголовочном файле с именем `eprintf.h`:

```
/* eprintf.h: функции, сообщающие об ошибках */
extern  void    eprintf(char *, ...);
extern  void    weprintf(char *, ...);
extern  char    *estrdup(char *);
extern  void    *emalloc(size_t);
extern  void    *erealloc(void *, size_t);
extern  char    *progname(void);
extern  void    setprogname(char *);
```

Этот заголовочный файл должен подключаться к любому файлу кода, в котором вызывается одна из перечисленных функций. Каждое сообщение об ошибке также может включать имя программы, если этого потребует вызывающий модуль. Имя устанавливается или извлекается тривиальными функциями `setprogname` и `progname`, объявленными в заголовочном файле и определенными в исходном коде модуля вместе с `eprintf`:

```
static char *name = NULL;  /* имя программы для сообщений */

/* setprogname: устанавливает заданное имя программы */
void setprogname(char *str)
{
    name = estrdup(str);
}

/* progname: возвращает хранимое имя программы */
char *progname(void)
{
    return name;
}
```

Типичное использование этих функций выглядит таким образом:

```
int main(int argc, char *argv[])
{
    setprogname("markov");
    ...
    f = fopen(argv[i], "r");
    if (f == NULL)
        eprintf("can't open %s:", argv[i]);
    ...
}
```

В результате выводится примерно следующее:

```
markov: can't open psalm.txt: No such file or directory
```

Эти вспомогательные функции оказались очень удобными при программировании некоторых наших задач, поскольку они унифицируют обработку ошибок, и само существование таких функций стимулирует скорее перехватывать ошибку, чем игнорировать ее. Тем не менее в них нет ничего особенного, и читатель может предпочесть вместо них какие-нибудь другие.

Предположим, что нам требуется не просто написать несколько функций для своего собственного употребления, а сформировать библиотеку для передачи другим пользователям. Что должна делать функция в такой библиотеке, если случается непоправимая ошибка? Функции, приведенные ранее в этой главе, выводили на экран сообщение об ошибке и завершали программу аварийно. Это вполне подходит для многих программ, особенно небольших утилит и автономных приложений. Но во многих случаях такой образ действий окажется неправильным, поскольку не даст другим модулям программы шанса исправить положение. Например, текстовый редактор должен попытаться не потерять информацию из редактируемого документа. В некоторых ситуациях библиотечная функция не должна даже выводить сообщение, потому что программа вполне может выполняться в среде, где такое со-

общение исказило бы выходные данные или просто пропало без следа. Разумный вариант — это выводить диагностические сообщения в файл-протокол, который можно читать независимо от других данных.

Перехватывайте ошибки на низком уровне, обрабатывайте их на высоком. Общий принцип гласит, что ошибки следует перехватывать на как можно более низком системном уровне, а вот обрабатывать их — на высоком. В большинстве случаев решать, что делать в связи с ошибкой, должен вызывающий, а не вызываемый модуль. Библиотечные функции могут помочь в этом, завершая ошибочную операцию корректно и неаварийно. Рассуждая таким образом, в случае несуществующего поля данных лучше возвращать `NULL`, чем завершать программу аварийно. Аналогично, функция `cvsgetline` возвращает `NULL` каждый раз независимо от того, сколько раз ее вызвали после обнаружения конца файла.

Надлежащие возвращаемые значения не всегда легко определить, как мы уже видели ранее при обсуждении вопроса о том, что должна возвращать функция `csvgetline`. Хотелось бы возвращать как можно больше полезной информации, причем в такой форме, какую легко было бы использовать в других модулях программы. В языках C, C++ и Java это означает возвращение некоего кодового значения оператором `return`, а фактических данных — по адресам (ссылкам) через аргументы. Во многих библиотечных функциях есть способ отличить возвращаемый нормальный результат от ошибочного. Так, функция ввода `getchar` возвращает элемент типа `char` в случае успеха и некоторое несимвольное значение типа `EOF` в случае конца файла или другой ошибки.

Этот механизм не срабатывает, если возможные корректные данные заполняют весь диапазон возвращаемых функцией значений. Например, математическая функция вычисления алгоритма `log` может возвращать любое вещественное значение. В библиотеке вещественной арифметики `IEEE` для индикации ошибки служит специальное значение `NaN` (“not a number” — “не число”), которое и возвращается в таких случаях.

В некоторых языках, например Perl или Tcl, имеются ненакладные способы организации групп из двух или более значений, именуемых *кортежами* (*tuples*). В таких языках значение функции и сигнал ошибки можно легко возвратить вместе. В библиотеке STL языка C++ есть тип данных `pair`, которым также можно воспользоваться в этих целях.

Было бы желательно отличать друг от друга различные исключительные состояния наподобие конца файла и др., а не сваливать их все в одну кучу и огульно называть “ошибками”. Если это трудно, можно ограничиться возвратом одного “исключительного” результата, но определить еще одну функцию, которая бы по требованию выдавала более подробную информацию о последней случившейся ошибке.

Именно такой подход принят в системе Unix и стандартной библиотеке C, в которой многие системные вызовы и библиотечные функции возвращают `-1`, а также помещают специфический код ошибки в глобальную переменную `errno`. Функция `strerror` служит для возврата строкового сообщения, ассоциированного с кодом ошибки. Рассмотрим следующую программу:

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
```

```
#include <math.h>

/* errno main: тестирует errno */
int main(void)
{
    double f;

    errno = 0;  /* сброс кода ошибки */
    f = log(-1.23);
    printf("%f %d %s\n", f, errno, strerror(errno));
    return 0;
}
```

В нашей системе она вывела следующее сообщение:

```
nan0x10000000 33 Domain error
```

Как показано выше, вначале переменную `errno` необходимо очистить (сделать равной нулю); затем в случае ошибки она получает ненулевое значение.

Используйте исключительные ситуации для действительно исключительных случаев. В некоторых языках для перехвата необычных ситуаций и принятия мер существует механизм *исключительных ситуаций*. Если случается ошибка, выполнение программы направляется по альтернативному пути. Исключительные ситуации не следует использовать для работы с ожидаемыми, нормальными значениями данных. Например, чтение данных из файла в конце концов приведет к тому, что встретится его конец. Эту ситуацию следует обрабатывать нормально, с применением возвращаемого из функции кода, а не перехвата исключительной ситуации.

В языке Java пишут так:

```
String fname = "someFileName";
try {
    FileInputStream in = new FileInputStream(fname);
    int c;
    while ((c = in.read()) != -1)
        System.out.print((char) c);
    in.close();
} catch (FileNotFoundException e) {
    System.err.println(fname + " not found");
} catch (IOException e) {
    System.err.println("IOException: " + e);
    e.printStackTrace();
}
```

В этом фрагменте кода в цикле считываются символы, пока не встретится конец файла. Сигнал об этом событии — возвращение значения `-1` из функции `read`. Но если файл не открывается, то возникает исключительная ситуация. В языках C и C++ в этом случае с входным потоком был бы ассоциирован указатель `null`. Наконец, если в блоке `try` случается еще какая-нибудь ошибка ввода-вывода, то снова возникает исключительная ситуация, которая обрабатывается в блоке `IOException`.

Исключительными ситуациями часто злоупотребляют. Из-за их способности изменять ход выполнения программы часто возникают весьма извилистые конструкции, сильно подверженные ошибкам. Неудача при открытии файла — это не очень-то экстраординарное событие; генерирование исключительной ситуации для такого случая выглядит слишком сильной мерой. Исключительные ситуации лучше

приберечь для действительно непредсказуемых событий типа переполнения файловых систем или ошибок вещественной арифметики.

В программах на С с помощью пары функций `setjmp` и `longjmp` можно организовать низкоуровневую системную базу для построения механизма исключительных ситуаций. Но эти функции достаточно сложны в применении, так что мы не будем здесь вдаваться в подробности.

А как насчет восстановления ресурсов в случае ошибки? Должна ли библиотека пытаться восстановить ресурсы, если случается ошибка? Обычно не должна, но она может оказать услугу другим модулям, постаравшись оставить после себя информацию в настолько корректном и безопасном состоянии, насколько это возможно. Гарантированно неиспользуемая память должна вернуться в систему. Если какие-то переменные остаются доступными программе, их значения следует сделать корректными. Обычным источником ошибок является обращение к указателю, который ссылается на освобожденную память. А вот если обработчик ошибок после освобождения памяти обнулит указатели, ссылающиеся на нее, то такое обращение не останется незамеченным. Функция `reset` во второй версии библиотеки CSV как раз реализовала попытку решения этих проблем. В целом нужно стремиться к тому, чтобы после случившейся ошибки библиотека осталась в работоспособном состоянии.

4.8. Пользовательские интерфейсы

До сих пор мы говорили об интерфейсах между компонентами одной программы или между программами. Но существует еще один важный интерфейс — между программой и ее пользователем-человеком.

Большинство примеров программ в этой книге снабжено довольно простыми консольно-текстовыми интерфейсами. Как говорилось в предыдущем разделе, необходимо стремиться к тому, чтобы распознавать ошибки, выводить сообщения о них и пытаться восстановить нормальный ход операций. Сообщения об ошибках должны содержать всю имеющуюся информацию о них и быть как можно более понятными без контекста. Например, следующее сообщение невразумительно:

```
estrdup failed
```

На самом деле ему положено выглядеть так:

```
markov: estrdup("Derrida") failed: Memory limit reached
```

Добавить эту дополнительную информацию в сообщение ничего не стоит, а пользователю от этого может быть большая польза при поиске источника ошибки или корректировке исходных данных.

В случае необходимости программы должны сообщать пользователю, как их правильно запускать. Это делается примерно такими функциями:

```
/* usage: выводит инструкцию и завершает программу */
void usage(void)
{
    fprintf(stderr, "usage: %s [-d] [-n nwords]"
        " [-s seed] [files ...]\n", progname());
    exit(2);
}
```

Имя программы указывает на источник сообщения, что особенно важно при работе с большим программным процессом. Если же программа выдает сообщение типа “синтаксическая ошибка” или “ошибка в такой-то функции”, то пользователь может вообще не понять, откуда это сообщение пришло.

В тексте сообщений об ошибках, командных приглашений, диалоговых окон должна содержаться информация о правильной форме исходных данных. Не пишите, что параметр слишком велик,— сообщите допустимый диапазон значений. Там, где это возможно, сам текст сообщения должен представлять собой корректные исходные данные — например, полную командную строку с правильными аргументами. Кроме того, что такие сообщения направляют пользователя, они еще позволяют скопировать данные в файл или буфер обмена для повторного запуска процесса. Отсюда виден недостаток диалоговых окон: их содержимое плохо поддается копированию и сохранению для последующего использования.

Один из эффективных способов организации хорошего пользовательского интерфейса состоит в том, чтобы разработать специализированный язык для задания параметров, выполнения операций и т.п. Хорошая система обозначений помогает сделать программу легкой в использовании, а также способствует ее качественной реализации. Интерфейсы на основе специализированных языков рассматриваются в главе 9.

Очень важно программировать безопасно и устойчиво — так, чтобы программа была неуязвимой по отношению к любым некорректным входным данным. Это важно и с позиций системной безопасности, и для защиты пользователя от самого себя. Более подробно этот вопрос рассматривается в главе 6, посвященной тестированию.

Для большинства людей при слове “интерфейс” перед глазами возникает графический интерфейс пользователя, характерный для современных компьютеров. Графические интерфейсы — это отдельная огромная тема для обсуждения, поэтому здесь мы скажем всего несколько слов. Во-первых, графические интерфейсы очень трудно разрабатывать и доводить до безупречно надежного состояния, поскольку их работа сильно зависит от поведения и желаний человека. Во-вторых, если у системы есть пользовательский интерфейс, то именно операции по его обслуживанию составляют бо́льшую часть кода, тогда как сами алгоритмы обработки данных занимают сравнительно скромный объем.

И тем не менее вышеизложенные принципы применимы как к внешнему дизайну программ, так и к внутренним реализациям пользовательских интерфейсов. Если подходить с позиций пользователя, то все качества хорошего стиля программирования, такие как простота, понятность, стандартизация, единообразие, идиоматичность, модульность, — это всего лишь шаги на пути к хорошему и легкому в применении интерфейсу. Отсутствие этих качеств обычно порождает неудобные в работе и неуклюжие интерфейсы.

В частности, весьма желательны единообразие и стандартизация интерфейса. Под этим подразумевается единая система понятий, единиц измерения, форматов, макетов, шрифтов, цветов, размеров и всех прочих характеристик, присущих графической системе. Например, сколько разных слов следует использовать в англоязычной программе для обозначения команды выхода из программы или закрытия окна? Их существует полтора десятка — от `Abandon` до <Ctrl+Z> включительно. Даже знаток английского языка может запутаться, не говоря уже об иностранце.

В графических системах интерфейсы играют особую роль, поскольку такие системы обычно велики, сложны в реализации и управляются совершенно другой моделью ввода данных, отличной от последовательного считывания текста. Великолепно проявили себя при программировании графических интерфейсов объектно-ориентированные языки, поскольку в них есть способы инкапсулировать в одном объекте состояние и возможные операции окна и т.п., а также собрать общие черты элементов интерфейса в базовых классах, вынеся отличия между ними в производные классы.

Дополнительная литература

Несмотря на устаревшие технические детали, книгу Frederick P. Brooks, Jr., *The Mythical Man Month* (Addison-Wesley, 1975; Anniversary Edition 1995) по-прежнему стоит читать; она содержит взгляды на разработку программного обеспечения, в той же мере ценные сегодня, как и во время ее первого издания.

Практически в каждой книге по программированию найдется какая-нибудь полезная информация о разработке интерфейсов. Одна из книг, John Lakos, *Large-Scale C++ Software Design* (Addison-Wesley, 1996), основана на приобретенном тяжкими трудами опыте и посвящена вопросам проектирования и реализации очень больших программ на C++. Хорошим изложением аналогичных вопросов по программированию на C является книга David Hanson, *C Interfaces and Implementations* (Addison-Wesley, 1997).

Превосходное описание групповой разработки программ с подчеркиванием важной роли чернового прототипирования дано в книге Steve McConnell, *Rapid Development* (Microsoft Press, 1996).

Имеется несколько интересных книг по разработке графических пользовательских интерфейсов, написанных с самых разных позиций. Мы предлагаем следующие источники из их числа:

Kevin Mullet, Darrell Sano, *Designing Visual Interfaces: Communication Oriented Techniques* (Prentice Hall 1995).

Ben Shneiderman, *Designing The User Interface: Strategies for Effective Human-Computer Interaction* (3rd edition, Addison-Wesley, 1997).

Alan Cooper, *About Face: The Essentials of User Interface Design* (IDG, 1995).

Harold Thimbleby, *User Interface Design* (Addison-Wesley, 1990).

5

Отладка

bug.

b. Дефект или изъян в машине, плане и т.п. *Досл.* “насекомое, клоп”. *Происх. америк.*

Pall Mall Gaz., 11 марта 1889 г., 1/1. Г-н Эдисон, по его словам, не сомкнул глаз две прошлые ночи, разыскивая “клопа” (“bug”) в своем фонографе. Данное выражение обозначает поиск неисправности; этим как бы подразумевается, что некое воображаемое насекомое спряталось внутри машины и является источником проблемы.

Оксфордский словарь английского языка, 2-е изд.
(Oxford English Dictionary)

В предыдущих четырех главах мы представили на суд читателя большой объем кода и при этом притворились, что все приведенные программы и функции заработали с первого же запуска. Конечно же, это не так — ошибок возникало достаточно много. Слово “bug” (“клоп”), употребляемое в программировании для обозначения ошибки, родилось не в среде программистов, а гораздо раньше, и тем не менее именно в этой индустрии оно является одним из самых распространенных. Так в каких же щелях программ прячутся эти “клопы”?

Общая причина ошибок состоит в разнообразии способов, с помощью которых могут взаимодействовать компоненты программ. Поскольку программы состоят именно из компонентов и взаимодействия между ними, это увеличивает сложность системы и повышает вероятность ошибок. Создано множество методов для ограничения взаимодействия компонентов, чтобы информация передавалась в программе по как можно меньшему количеству каналов. Среди таких методов — сокрытие данных, абстрагирование, деление на интерфейс и реализацию, различные специфические языковые средства. Существуют также методы для контроля целостности и согласованности программных архитектур — анализ спецификаций, формализованные процедуры верификации, различные техники моделирования. Но все это не затронуло фундаментальные основы индустрии разработки программ; сколько-нибудь принципиальное отличие видно только на примере небольших задач. Реальность такова, что ошибки есть и будут; их обнаруживают тестированием, а устраняют отладкой.

Хорошие программисты знают, что на отладку уходит столько же времени, сколько и на написание кода, поэтому они стараются учиться на своих ошибках.

Каждая найденная ошибка научит вас либо не делать ее снова, либо обнаружить ее в кратчайшие сроки, если она все-таки случится.

Отладка — дело нелегкое, и отнимает она непредсказуемое количество времени. Поэтому следует поставить перед собой такую цель — тратить на отладку как можно меньше времени. Для этого есть ряд методов и подходов: качественное проектирование, хороший стиль, проверка граничных условий и предельных случаев, вставка контрольных условий в код, стремление к устойчивости по отношению к входным данным, хорошо определенные интерфейсы, минимальное использование глобальных данных, привлечение средств диагностики. Как известно, профилактика обходится гораздо дешевле лечения.

Какова роль языка программирования во всем этом? Главная движущая сила в развитии языков программирования — это как раз стремление предотвратить возможность ошибки средствами самих языков. Некоторые средства и свойства языков действительно делают определенные классы ошибок гораздо менее вероятными. К таким средствам относится контроль диапазона индексов массивов, ограниченное использование указателей или вообще отказ от них, сборка мусора, специальные строковые типы данных, типизированный ввод-вывод, строгие требования к соответствию типов. Есть и такие средства, которые особенно сильно провоцируют возникновение ошибок. Это оператор `goto`, глобальные переменные, неконтролируемые указатели, автоматические преобразования типов. Программистам следует знать слабые или рискованные места своих языков и соблюдать особую осторожность при их применении. Необходимо также включать все проверки, поддерживаемые компилятором, и обращать внимание на все предупреждения.

Каждое средство, предотвращающее возможность той или иной ошибки, имеет свою цену. Если, например, язык высокого уровня не позволяет сделать простую низкоуровневую ошибку, это означает, что в нем можно наделать ошибок более высокого уровня. Никакой язык не может гарантировать отсутствия ошибок.

Как бы мы ни протестовали против такой перспективы, большая часть времени программистов все равно уходит на тестирование и отладку. В этой главе рассматривается вопрос о том, как сделать процесс отладки как можно короче и продуктивнее. К тестированию мы вернемся в главе 6.

5.1. Отладчики

Компиляторы основных языков программирования обычно поставляются вместе с многофункциональными отладчиками, нередко в виде единой среды разработки, в которой интегрируются все операции по созданию и редактированию исходного кода, компиляции, выполнению и отладке. Отладчики часто имеют графический интерфейс для пошагового выполнения операторов или функций программы; они способны приостанавливать выполнение на заданных строках или при выполнении заданного условия. В них также имеются средства форматирования и отображения значений переменных.

Отладчик можно активизировать непосредственно тогда, когда становится очевидным наличие проблемы. Некоторые отладчики берут власть в свои руки в тот момент, когда при выполнении программы происходит что-то неожиданное. Обычно

бывает нетрудно выяснить, в каком месте прервалось выполнение программы, проследить последовательность активных в тот момент функций (*стековый фрейм*) и отобразить значения локальных и глобальных переменных. Этой информации может оказаться вполне достаточно для обнаружения ошибки. Если же это не так, то с помощью точек останова и пошагового выполнения можно выполнить аварийную программу в отладочном режиме и найти место первого проявления проблемы.

В подходящей среде и в руках квалифицированного программиста хороший отладчик может сделать отладку быстрой и эффективной, а то и вовсе тривиальной. Если существуют такие мощные инструменты отладки, зачем кому-нибудь понадобится обходиться без них? Чего ради, спросит читатель, мы написали целую главу, посвященную отладке?

Для этого имеется ряд причин. Некоторые из них вполне объективны, а другие основаны на нашем личном опыте. Некоторые языки, не относящиеся к числу наиболее распространенных, не имеют отладчиков или предоставляют только зачаточные возможности отладки. Отладчики привязаны к операционной системе, поэтому при переходе из одной среды в другую можно не найти знакомого отладчика. Некоторые программы плохо поддаются отладке с помощью отладчика; среди таковых — многопоточные приложения, параллельные процессы, операционные системы, распределенные сетевые системы. Все эти классы программ часто приходится отлаживать на более низком уровне, и в этом случае вы остаетесь наедине с программой без всякой помощи, кроме разве что дополнительных операторов вывода, вооруженные только своим опытом и навыками анализа кода.

На наш собственный вкус, отладчики лучше использовать только для получения стекового фрейма и значений одной-двух переменных. Одна из причин нашей нелюбви к отладчикам состоит в том, что бывает очень легко заблудиться в подробностях сложных структур данных и управляющих конструкций. Пошаговое выполнение программы, по нашему мнению, менее продуктивно, чем тщательное ее обдумывание и добавление операторов вывода или дополнительных проверок в критических местах кода. Перебор всех операторов по очереди отнимает больше времени, чем просмотр результатов хорошо продуманного отладочного вывода. Можно гораздо быстрее решить, куда вставить дополнительную проверку или контрольный вывод, чем “дошагать” до критического фрагмента кода, даже зная, где он находится. Что еще более важно, контрольные операторы остаются в тексте программы, а сеансы отладки неизбежно заканчиваются.

“Слепое зондирование” с помощью отладчика далеко не всегда бывает продуктивным. Гораздо полезнее выяснить с помощью отладчика, в каком состоянии находилась программа в момент аварийного завершения, а затем подумать, как это состояние могло наступить. Пользоваться отладчиками нелегко, и их поведение часто ставит в тупик. Начинающим программистам они вообще скорее помеха, чем подспорье. Если вы задали отладчику неправильный вопрос, он, вероятнее всего, что-то ответит, но эта информация может завести вас в дебри.

И все же отладчики могут принести огромную пользу, поэтому такой инструмент следует держать наготове в своем арсенале. Часто в случае возникновения ошибки именно он первым приходит на помощь. Но если у вас нет отладчика или вы застряли на особенно сложной проблеме, прибегните к методам и подходам, изложенным далее в этой главе, и убедитесь, что в любом случае отладку можно

выполнить быстро и эффективно. Эти подходы помогут также и при работе с отладчиком, поскольку касаются общих принципов того, как разобраться в ошибках и их вероятных причинах.

5.2. Простые ошибки с очевидными симптомами

Оп-ля! Случилась ошибка. Программа завершилась аварийно, или вывела на экран какую-то нелепицу, или же зациклилась и выполняется без остановки. Что же делать?

Новички склонны обвинять в ошибке компилятор, библиотеки и вообще все, что угодно, кроме их собственного кода. Опытным программистам тоже хотелось бы так думать, но они не строят иллюзий и знают, что большинство проблем они создают себе сами.

К счастью, большинство ошибок достаточно просты и могут быть обнаружены *простыми методами*. Внимательно прочтите полученные на выходе ошибочные данные и проанализируйте, как они могли быть сгенерированы. Просмотрите отладочные выходные данные, выведенные непосредственно перед аварийным завершением; если возможно, получите стековый фрейм с помощью отладчика. Теперь вам уже известно кое-что о том, что именно случилось и где. Сделайте паузу и подумайте. Как это могло случиться? Мысленно проследите ход работы программы назад от места ее аварии, выдвигая предположения о возможной причине.

Отладка программы, подобно раскрытию преступления, всегда подразумевает дедуктивное рассуждение. Случилось нечто невероятное, и единственная достоверная информация об этом — сам факт, что оно произошло. Поэтому необходимо думать в обратном направлении от события, чтобы попытаться понять его причины. Имея исчерпывающее объяснение, мы будем знать, какие меры принять. Заодно, скорее всего, выяснится еще пара вещей, которых мы сами не ожидали обнаружить.

Ищите знакомые конструкции. Спросите себя, знакома ли вам та или иная программная конструкция. Мысль "где-то я это уже видел" представляет собой первый шаг к пониманию, а иногда и дает нужный ответ. Самые распространенные ошибки имеют свои неповторимые проявления. Например, начинающие программисты на С часто пишут так:

```
?    int n;
?    scanf("%d", n);
```

Это неправильно. Необходимо писать так:

```
     int n;
     scanf("%d", &n);
```

В результате этой ошибки, как правило, предпринимается попытка обращения к недоступному участку памяти. Преподаватели языка С распознают симптомы этой ошибки мгновенно.

Бесконечным источником простых ошибок является несоответствие типов и их преобразование в функциях `printf` и `scanf`:

```
?    int n = 1;
?    double d = PI;
?    printf("%d %f\n", d, n);
```

Симптомом этой ошибки часто является вывод на экран нелепых значений переменных: огромных целых чисел, нереально маленьких или больших вещественных. В системе Sun SPARC эта программа выдала одно большое число и одно — просто астрономическое (чтобы оно поместилось на страницу, пришлось разбить его на несколько строк):

```
1074340347 268156158598852001534108794260233396350\
    1936585971793218047714963795307788611480564140\
    0796821289594743537151163524101175474084764156\
    422771408323839623430144.000000
```

Еще одна распространенная ошибка — это употребление спецификации `%f` вместо `%lf` для ввода чисел типа `double` с помощью функции `scanf`. Некоторые компиляторы распознают такие ошибки, проверяя соответствие между типами аргументов `printf` и `scanf` и соответствующими спецификациями в строках формата. Если активизировать все предупреждающие сообщения, то для такого вызова `printf`, какой бы показан выше, компилятор `gcc` выведет на экран следующее:

```
x.c:9: warning: int format, double arg (arg 2)
x.c:9: warning: double format, different type arg (arg 3)
```

Характерные ошибки возникают также, если не инициализировать локальную переменную. В результате можно получить очень большое число — "мусор", оставшийся в данной ячейке памяти от предыдущих переменных. Некоторые компиляторы предупреждают о подобных ситуациях, но для этого часто приходится включать специальную проверку при компиляции, и все равно некоторые случаи ускользают от внимания компилятора. Участки памяти, выделенные функциями `malloc`, `realloc` и т.п., также часто содержат "мусор". Не забывайте инициализировать такие буферы памяти.

Начните проверку с последнего изменения. Итак, какое же последнее изменение вы внесли в программу? Если по мере доработки программы вы вносили постепенные изменения, по одному за раз, то либо источник ошибки находится в новом фрагменте кода, либо новый код способствует ее проявлению. Чтобы локализовать проблему, тщательно проанализируйте последние изменения. Если в новой версии ошибка есть, а в старой — нет, следовательно, новый код имеет отношение к проблеме. Это означает, что нужно как минимум всегда хранить предыдущую версию программы, в правильности которой вы более или менее уверены, чтобы было с чем сравнивать. Необходимо также вести учет сделанных изменений и исправленных ошибок, чтобы не пришлось заново разыскивать эту жизненно важную информацию, одновременно пытаясь решить проблему. Здесь существенную помощь могут оказать различные механизмы фиксации истории изменений, такие как системы управления версиями.

Не делайте одну и ту же ошибку дважды. После исправления ошибки задайте себе вопрос, не могли ли вы сделать такую же в другом месте. Именно это случилось с одним из авторов за несколько дней до написания этой главы. Программа представляла собой черновой прототип для коллеги и содержала шаблонную конструкцию для обработки необязательных аргументов-ключей:

```
?    for (i = 1; i < argc; i++) {
?        if (argv[i][0] != '-')  /* ключи закончились */
?            break;
?        switch (argv[i][1]) {
?        case 'o':               /* выходной файл */
?            outname = argv[i];
?            break;
?        case 'f':
?            from = atoi(argv[i]);
?            break;
?        case 't':
?            to = atoi(argv[i]);
?            break;
?        ...
```

Как только наш коллега испробовал эту программу, он сообщил, что имя выходного файла результатов неизменно получало префикс `-o`. Это было неприятно, но легко поправимо; следовало лишь написать так:

```
    outname = &argv[i][2];
```

Исправление было внесено, и программа снова отправлена на выполнение. Тут же пришел ответ, что программе не удается корректно обработать такой аргумент, как `-f123`; преобразованное из строки в число значение всегда равнялось нулю. Это ошибка того же типа; следующий блок `case` в операторе `switch` должен был выглядеть так:

```
    from = atoi(&argv[i][2]);
```

Поскольку автор работал в спешке, он не заметил, что та же самая ошибка попадалась еще в двух местах, так что потребовалось дополнительное время на исправление всех принципиально схожих ошибок.

В несложных фрагментах кода ошибки появляются тогда, когда мы теряем бдительность из-за излишне знакомых конструкций. Даже если код настолько прост, что его, казалось бы, можно писать во сне, все равно не делайте этого — держите глаза широко открытыми.

Не откладывайте отладку. Торопливость создает проблемы во многих ситуациях. Не игнорируйте аварийное завершение программы — проанализируйте его причины немедленно, потому что позже оно может и не случиться, а потом будет слишком поздно. В качестве знаменитого примера можно привести проект “Mars Pathfinder”. После безупречного приземления на Марсе в июле 1997 года компьютеры этого автоматического космического корабля начали перезагружаться каждые один-два дня, что совершенно озадачило инженеров. Проанализировав источник проблемы, они вдруг поняли, что уже встречались с ней раньше. Такие перезагрузки случались и в предполетных испытаниях, но их попросту игнорировали, потому что

велись более важные работы над другими проблемами. Итак, решение задачи было отложено на более позднее время, а когда это время наступило, компьютеры уже находились за десятки миллионов километров от Земли, и отладка стала, мягко говоря, затруднительной.

Анализируйте стековый фрейм. Хотя с помощью отладчиков можно успешно следить за выполняющейся программой, наиболее полезное их применение — это изучение состояния программы после ее аварийного завершения. Самой полезной частью выдаваемой отладчиком информации можно считать номер строки, на которой программа прекратила работу; этот номер часто фигурирует в стековом фрейме. Большим подспорьем служат также ошибочные значения данных программы (нулевые указатели; огромные целые числа, которым следовало бы быть маленькими, или отрицательные числа, которым полагалось быть положительными; неалфавитные символы).

Приведем типичный пример, основываясь на нашей теме сортировки из главы 2. Чтобы отсортировать массив целых чисел, следует вызвать функцию `qsort`, передав в нее функцию сравнения целых чисел `icmp`:

```
    int arr[N];
    qsort(arr, N, sizeof(arr[0]), icmp);
```

А теперь представим себе, что в эту функцию случайно передали имя функции сравнения строк `scmp`:

```
?   int arr[N];
?   qsort(arr, N, sizeof(arr[0]), scmp);
```

В данном случае компилятор не сможет распознать никакого несоответствия типов, так что крах неизбежен. При выполнении программы происходит ее аварийное завершение, поскольку она делает попытку обратиться к адресу памяти, защищенному от доступа. Отладчик `dbx` дает следующий стековый фрейм (текст отформатирован для удобства):

```
    0 strcmp(0x1a2, 0x1c2) ["strcmp.s":31]
    1 scmp(p1 = 0x10001048, p2 = 0x1000105c) ["badqs.c":13]
    2 qst(0x10001048, 0x10001074, 0x400b20, 0x4) ["qsort.c":147]
    3 qsort(0x10001048, 0x1c2, 0x4, 0x400b20) ["qsort.c":63]
    4 main() ["badqs.c":45]
    5 __istart() ["crt1tinit.s":13]
```

Отсюда видно, что программа потерпела крах при выполнении функции `strcmp`. Анализ значений аргументов показывает, что два указателя, переданные в `strcmp`, слишком малы, а это верный признак ошибки. Стековый фрейм содержит последовательность номеров строк, в которых вызывались функции. Строка 13 в файле `badqs.c` соответствует следующему вызову:

```
    return strcmp(v1, v2);
```

Очевидно, именно этот вызов оказался аварийным, так что отсюда и следует исходить при поисках ошибки.

С помощью отладчика можно также получить значения глобальных или локальных переменных в определенный момент, что также помогает при выяснении обстоятельств аварии.

Сначала читайте, потом исправляйте. Эффективный, хотя и недооцененный способ отладки — это тщательный анализ кода и его глубокое обдумывание до внесения каких-либо исправлений. При появлении ошибки у программиста неизменно рождается могучий порыв броситься к клавиатуре и начать исправлять программу, чтобы ошибка поскорее исчезла. Но вполне возможно, что он попросту не понимает истинной природы ошибки и исправит что-нибудь не то, в результате наделав новых ошибок. Чтобы взглянуть на код под другим углом, чем на экране компьютера, распечатайте критический фрагмент программы на бумаге. К тому же это поощряет думать над кодом медленнее и тщательнее. И все же не нужно делать из распечатки культа. Может получиться так, что загубив уйму бумаги, вы потом попросту не сможете как следует проанализировать структуру программы, разбросанную по множеству страниц. Кроме того, выведенный на печать листинг программы устареет в тот самый момент, когда вы начнете вносить исправления.

Если ничего не получается, сделайте перерыв. Иногда вы видите в исходном коде не то, что там написано, а то, что вам хотелось бы видеть. Перерыв даст вам возможность успокоиться и вернуть себе объективность в восприятии кода. Итак, сопротивляйтесь искушению немедленно что-то набирать; разумная альтернатива — это как следует подумать над кодом.

Объясните код кому-то другому. Объяснение своего исходного кода кому-то другому — это очень эффективный метод. По сути, чаще всего вы объясняете его себе самому. Иногда требуется всего лишь несколько фраз, после которых вас осеняет догадка, и остается только сказать коллеге: “Ой, я уже все понял. Извини за беспокойство”. Этот подход исключительно эффективен даже тогда, когда слушатель не является программистом. В одном из университетских вычислительных центров недалеко от рабочего стола консультанта восседал плюшевый медведь. Студенты, озадаченные таинственными ошибками в своих программах, обязаны были сперва поведать свои беды медведю, и только после этого им разрешалось обратиться за советом к преподавателю.

5.3. Сложные ошибки с трудными симптомами

“Ничего не понятно. Что же делать?” Если при анализе ошибки у вас возникает только эта мысль, значит, жизнь всерьез зашла в тупик.

Сделайте ошибку воспроизводимой. Первый шаг к решению — это заставить ошибку проявляться по своему желанию. Нет ничего хуже, чем гоняться за ошибкой, которая еще и проявляется не всякий раз. Потратьте некоторое время и сконструируйте такие входные данные и наборы параметров, которые гарантированно будут вызывать проблему к жизни. Затем оформите запуск программы так, чтобы его можно было выполнять щелчком на кнопке или несколькими нажатиями клавиш.

Если ошибка действительно серьезна, вам придется прогонять ее в процессе отладки снова и снова, а значит, необходимо иметь удобный способ делать это.

Если ошибку не удается проявить каждый раз по своему желанию, подумайте, почему это так. Какой набор условий заставляет ее появляться чаще — хотя бы с большей вероятностью, — чем остальные? Даже если не удается увидеть ошибку каждый раз, все же можно сэкономить много времени, сократив срок ее ожидания.

Если программа может выдавать отладочные выходные данные, активизируйте эту возможность. Очень желательно, чтобы модельные программы наподобие алгоритма генерирования текста по марковским цепям (см. главу 3) предоставляли отладочную информацию по требованию программиста. Это может быть, например, инициализатор датчика случайных чисел, по которому можно восстановить генерируемые выходные данные. Также желательно, чтобы и сам инициализатор можно было изменить по мере необходимости. Подобные возможности есть во многих программах, и в ваши собственные также стоит включить аналогичные средства.

Применяйте подход "разделяй и властвуй". Подумайте, можно ли сделать концентрированную "выжимку" входных данных, из-за которых программа завершается аварийно. Сузьте круг подозрений, построив минимально возможный набор исходных данных из тех, которые вызывают ошибку к жизни. Выясните, какие изменения в этих данных заставляют ошибку исчезнуть. Попытайтесь найти как можно более точные тестовые примеры для проявления ошибки. Каждый тестовый пример должен давать определенный результат, который подтверждает или отпровергает ту или иную гипотезу относительно природы ошибки.

Затем примените поиск методом деления пополам. Отбросьте половину входных данных и посмотрите, по-прежнему ли на выходе получается ошибочный результат. Если нет, то вернитесь к предыдущему состоянию и отбросьте другую половину данных. Та же процедура применима и к самому тексту программы: уберите часть программы, которая предположительно не имеет отношения к ошибке, и проверьте, не исчезла ли ошибка. Для фрагментирования больших тестовых данных и текстов программ без потери ошибки удобно пользоваться редактором с командой отмены последней операции (Undo).

Анализируйте числовые закономерности, присущие ошибкам. Иногда удается сузить поле исследований, заметив определенную закономерность в статистических параметрах данных, которые вызвали ошибку. Например, мы с удивлением обнаружили несколько орфографических ошибок в одном из свеженаписанных разделов этой книги. Некоторые буквы попросту исчезли из текста. Это поставило нас в тупик. Текст создавался путем копирования и вставки фрагментов из другого файла, поэтому естественно было предположить, что ошибка кроется в реализации команд для вырезания и вставки в текстовом редакторе. Но с чего начать поиск проблемы? Мы поискали подсказку в самом тексте, и нам показалось, что пропущенные символы довольно равномерно разбросаны по тексту. Мы измерили интервалы между ними и обнаружили, что расстояние между соседними пропущенными символами составляло в точности 1023 байта. Это число слишком знакомо выглядит, чтобы быть случайным. Поиск чисел, близких к 1024, в исходном коде редактора дал несколько потенциально ошибочных мест. Одно из них находилось в новом, недавно добавленном коде; мы проверили его первым и легко обнаружили

классическую ошибку пропуска одной позиции — нулевой байт, т.е. символ конца строки, затирал последний значащий символ в 1024-символьном буфере.

Итак, изучение числовых характеристик выходных данных вывело нас прямо на ошибку. Сколько это заняло времени? Пару минут мы пребывали в озадаченном состоянии, потом минут пять искали закономерность в расположении пропущенных символов, минуту-другую разыскивали потенциально ошибочные места в коде редактора и еще минуту устраняли саму ошибку. Нечего было и надеяться проделать это с помощью отладчика, поскольку в работе участвовали две многозадачные программы, управляемые командами мыши и обменивающиеся данными через файловую систему.

Выводите побольше данных на экран, чтобы локализовать поиск. Если толком непонятно, что делает программа, добавьте в нее операторы вывода, чтобы она отображала побольше данных. Это один из самых простых методов отладки, требующий минимальных трудозатрат. Вставляйте операторы так, чтобы проверять свое понимание работы программы или уточнять гипотезы о природе ошибок. Например, если вы уверены, что в какое-то место программы управление попасть не должно, вставьте в него оператор для вывода сообщения типа "сюда попасть нельзя". Если это сообщение появится на экране, передвиньте оператор назад по тексту программы, чтобы выяснить, в каком же месте события приняли неожиданный оборот. Или же сделайте наоборот: расставьте сообщения вида "выполняется то", "выполняется это" последовательно по тексту программы, чтобы потом обнаружить, в каком месте программа еще работала нормально. Каждое сообщение должно быть характерным, чтобы их легко можно было различать.

Выводите сообщения в компактной фиксированной форме, чтобы легко распознавать их на глаз или с помощью программ наподобие `grep` — средств поиска заданных образцов в текстах. (Иметь такую программу, как `grep`, при работе с текстами просто необходимо. В главе 9 приведена одна из простых реализаций подобного алгоритма.) Если отображается значение переменной, то формат всякий раз должен быть одним и тем же. В языках C и C++ следует выводить указатели в виде шестнадцатеричных чисел по спецификации `%x` или `%p`; таким образом можно увидеть, равны ли между собой (или соотносятся иным образом) те или иные указатели. Научитесь правильно читать значения указателей, различать правдоподобные и неправдоподобные их значения типа нуля, отрицательных, нечетных или слишком малых чисел. Знакомство с формой представления адресов также пригодится при работе с отладчиком.

Если программа имеет тенденцию выводить слишком много данных, может оказаться достаточным выводить контрольные сообщения о прохождении тех или иных точек программы в виде всего одной буквы: `A`, `B`, `C` и т.д.

Пишите код с самопроверкой. Если требуется более подробная информация, напишите свою собственную функцию для проверки того или иного условия, контрольного вывода значений переменных и аварийного завершения программы:

```
/* check: проверка условия, контрольный вывод, завершение */
void check(char *s)
{
    if (var1 > var2) {
```

```
            printf("%s: var1 %d var2 %d\n", s, var1, var2);
            fflush(stdout); /* завершаем операции вывода */
            abort();        /* сигнализируем аварийное завершение */
        }
    }
```

Мы написали функцию `check`, чтобы с ее помощью вызывать функцию `abort` из стандартной библиотеки С, которая вызывает аварийное завершение выполнения программы для последующего анализа при помощи отладчика. В каком-нибудь другом случае, возможно, понадобится продолжить выполнение программы после вывода контрольных данных.

Функцию `check` или ей подобную следует вставлять в код всякий раз, когда она может оказаться уместной:

```
check("before suspect");
/* ... подозрительный код ... */
check("after suspect");
```

После исправления ошибки не стоит просто выбрасывать функцию `check` из исходного кода. Оставьте ее в тексте, закомментировав или поместив в условно компилируемый блок, чтобы при появлении следующей трудной ошибки можно было легко восстановить вызовы функции.

Для более тяжелых случаев функцию `check` можно переделать так, чтобы она выполняла сложные проверки и вывод структур данных. Этот подход можно обобщить, построив систему функций для текущих проверок структур данных и другой рабочей информации по ходу выполнения программы. Если в программе используются очень сложные структуры данных, то такие проверочные функции неплохо написать заранее — *до того*, как случится катастрофа. Их следует систематически спроектировать и реализовать вместе с программой, а затем по необходимости активизировать. Пользуйтесь ими не только при отладке; пусть они присутствуют в программе на всех этапах ее разработки. Если это не сильно отражается на быстродействии, то полезно даже оставить их всегда активными. Большие программы наподобие систем телефонной коммутации нередко содержат "ревизионные" подсистемы довольно значительного объема, задача которых состоит в контроле состояния информации и оборудования, а также в обнаружении и исправлении возникающих ошибок.

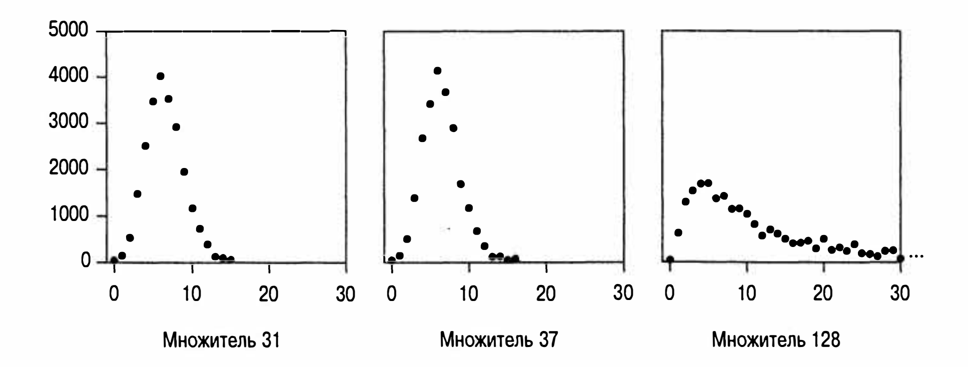
Записывайте файл протокола. Одним из надежных методов отладки является протоколирование выполнения программы, т.е. запись потока служебной информации в файл в каком-либо фиксированном формате. Если случается авария, в таком протоколе можно найти отчет о том, что произошло непосредственно перед аварийным завершением. Web-серверы и другие подобные сетевые программы ведут обширные протоколы сетевых транзакций, отслеживая все операции как своих клиентов, так и самих себя. Вот выдержка из протокола одной из локальных систем:

```
[Sun Dec 27 16:19:24 1998]
HTTPd: access to /usr/local/httpd/cgi-bin/test.html
    failed for m1.cs.bell-labs.com
    reason: client denied by server (CGI non-executable)
    from http://m2.cs.bell-labs.com/cgi-bin/test.pl
```

Не забывайте очищать буферы ввода-вывода, чтобы в протокол попадала самая последняя информация. Функции вывода наподобие `printf` обычно буферизуют выводимые данные, чтобы сделать вывод эффективнее; при аварийном завершении программы буферизованные данные могут пропасть. В языке С вызов функции `fflush` гарантирует очистку буферов и запись данных перед завершением программы; в С++ и Java имеются аналогичные функции `flush` для потоков вывода. Или же, если вы можете позволить себе некоторое уменьшение быстродействия, проблему буферизации можно решить, отказавшись от нее при записи протоколов. Буферизация управляется стандартными функциями `setbuf` и `setvbuf`; оператор `setbuf(fp, NULL)` отключает буферизацию потока `fp`. Обычно по умолчанию потоки ошибок (`stderr`, `cerr`, `System.err`) не буферизуются.

Стройте графики. Часто графики и схемы приносят больше пользы при отладке, чем текстовые данные. Разного рода графические схемы особенно полезны при анализе структур данных (см. главу 2), не говоря уже о программах с графическим интерфейсом. Но и в других программах этот подход может пригодиться. Точечные диаграммы гораздо лучше показывают отклонения данных от нормы, чем столбцы цифр. Элементарная гистограмма данных позволяет обнаружить ненормальные отклонения в сводках экзаменационных оценок, наборах псевдослучайных чисел, размерах ячеек в хэш-таблицах и т.п.

Если вам непонятно, что происходит в вашей программе, попробуйте снабдить структуры данных статистикой и вычертить ее графики. Ниже приведены графики для программы `markov` из главы 3 (версия на языке С). На них по оси x отложена длина цепочек, а по оси y — количество элементов в цепочках данной длины. Здесь применяются наши стандартные входные данные в виде текста Псалтыри (42 685 слов, 22 482 префикса). Первые два графика соответствуют двум благоприятным значениям хэш-множителей, 31 и 37, а третий — неблагоприятному множителю 128. В первых двух случаях ни одна цепочка не имеет длину больше 15-16 элементов, причем большинство элементов организованы в цепочки длиной 5-6 элементов. В третьем случае распределение гораздо шире, самая длинная цепочка включает 187 элементов, и целые тысячи элементов содержатся в цепочках длиннее 20.



Пользуйтесь дополнительными программными средствами. Постарайтесь максимально эффективно воспользоваться программными средствами, имеющимися в той среде, где производится отладка. Например, существует программа `diff`, способная сравнить выходные данные успешного и аварийного сеансов отладки и тем самым дать возможность сконцентрироваться на отличиях между ними. Если выходные данные слишком объемисты, воспользуйтесь программой `grep` для поиска нужных элементов или текстовым редактором для подробного изучения этих данных. Не поддавайтесь искушению вывести выходные данные на печать: компьютеры способны отфильтровать большой объем данных лучше, чем это сделает человек. Используйте сценарии на командных языках и другие подобные средства для автоматизации обработки данных, полученных в ходе сеансов отладки.

Для проверки гипотез или своего понимания работы тех или иных языковых средств бывает полезно написать тривиальную программу. Например, допускается ли освобождать нулевой указатель? Для ответа на этот вопрос скомпилируйте и выполните следующую программу:

```
int main(void)
{
    free(NULL);
    return 0;
}
```

Программы контроля изменений в исходном коде, такие как RCS, помогают вести учет версий кода. С их помощью можно выяснить, какие изменения вносились в программу, и вернуться к одной из предыдущих версий, тем самым восстановив знакомое состояние. Кроме индикации недавних изменений, эти системы выполняют и другую важную функцию: они помогают выявить фрагменты кода, давно и часто подвергающиеся исправлениям. Именно в таких фрагментах чаще всего скрываются ошибки.

Записывайте все, что вы делаете. Если поиск ошибки продолжается бесконечно долго, можно легко потерять счет опробованным методам и сделанным выводам. Если же записывать проделанные тесты и полученные результаты, то вероятность пропустить что-то или ошибочно решить, что какой-то способ вы уже пробовали, резко снижается. Ведение записей помогает лучше запомнить суть проблемы на тот случай, если она или ей подобная возникнет где-нибудь в следующий раз; сделанные записи также помогут объяснить суть проблемы кому-то другому.

5.4. Тяжелые случаи

Что делать, если все приведенные выше советы оказались бесполезными? Наверное, пора взять хороший отладчик и выполнить программу шаг за шагом. Если ваше представление о работе программы не соответствует действительности, т.е. вы просто смотрите не в то место или не видите проблему, глядя на нее в упор, то отладчик заставит вас думать по-другому. Ошибки, связанные с неправильным представлением о программе, относятся к самым сложным; применение средств автоматизации в этом случае просто необходимо.

Иногда ошибочное представление имеет простой характер: незнание приоритета операций, неправильная конструкция оператора, отступы, не соответствующие структуре кода, конфликт области действия, при котором локальное имя превалирует над глобальным или глобальное вмешивается в локальную область видимости переменных. Например, программисты часто забывают, что знаки операций `&` и `|` имеют более низкий приоритет, чем `==` и `!=`, и пишут так:

```
?    if (x & 1 == 0)
?        ...
```

Выражение в скобках всегда ложно, поэтому блок никогда не выполняется, а программист этого не осознает. Иногда из-за простой опечатки одинарный знак равенства (=) превращается в двойной или наоборот:

```
?    while ((c == getchar()) != EOF)
?        if (c = '\n')
?            break;
```

Бывает, что из-за крохотной опечатки за бортом остается часть важных операций:

```
?    for (i = 0; i < n; i++);
?        a[i++] = 0;
```

Множество ошибок появилось на свет из-за поспешного набора текста:

```
?    switch (c) {
?        case '<':
?            mode = LESS;
?            break;
?        case '>':
?            mode = GREATER;
?            break;
?        defualt:
?            mode = EQUAL;
?            break;
?    }
```

Иногда ошибка возникает из-за передачи аргументов в неправильном порядке, а контроль соответствия типов ничем помочь не может:

```
?    memset(p, n, 0);    /* поместить n нулей в p */
```

Следовало бы записать это так:

```
?    memset(p, 0, n);    /* поместить n нулей в p */
```

Нередко нежелательные изменения происходят за вашей спиной: модифицируется глобальная переменная или ресурс общего пользования, а вы даже не знаете, какие именно модули к ним обращаются и с какими целями.

Бывает, что алгоритм или структура данных имеет внутренний дефект, а вы его не видите. Подготавливая материал по связанным спискам, мы написали пакет функций для работы со списками: создание новых элементов, добавление их в голову или хвост списка и т.д. (см. главу 2). Разумеется, мы написали и тестовую программу для проверки правильности этих функций. Первые несколько тестов

прошли успешно, а затем произошел крах. Тестовая программа была, в сущности, совсем простой:

```
?    while (scanf("%s %d", name, &value) != EOF) {
?        p = newitem(name, value);
?        list1 = addfront(list1, p);
?        list2 = addend(list2, p);
?    }
?    for (p = list1; p != NULL; p = p->next)
?        printf("%s %d\n", p->name, p->value)
```

Найти ошибку оказалось удивительно нелегким делом. Мы совершенно не осознавали, что в первом цикле один и тот же узел помещался в оба списка таким образом, что к моменту начала вывода указатели были безнадежно перепутаны.

Такие ошибки найти особенно трудно, потому что мозг отказывается заставить посмотреть на них в упор и пытается обойти. Здесь большую помощь оказывает отладчик, поскольку он заставляет мышление повернуть на правильный курс и следовать за фактическим ходом программы, а не за собственными домыслами. Часто проблема бывает связана со структурой всей программы, и для обнаружения ошибки приходится начинать с анализа исходных допущений.

Кстати, заметьте, что в примере со списками ошибка скрывалась в коде тестовой программы и поэтому ее было особенно трудно найти. Очень легко увлечься поиском ошибок там, где их нет, просто потому, что код для тестирования написан неправильно, или тестируется не та версия программы, или перед тестированием не выполняется перекомпиляция и обновление исходных данных.

Если вы потратили много труда и не нашли ошибку, сделайте перерыв. Отвлекитесь от задачи, займитесь чем-то другим. Поговорите с товарищем, попросите его о помощи. После этого ответ может прийти сам собой как гром с ясного неба, но даже если нет, по крайней мере во время следующего сеанса отладки вы уже не застрянете в той же колее.

В отдельных редких случаях проблема бывает действительно связана с компилятором, библиотеками, операционной системой или даже с аппаратными компонентами компьютера, особенно если перед самым появлением ошибки в рабочей среде были сделаны какие-то изменения. Никогда не следует начинать поиск ошибки с обвинения системных средств, но если все другие подходы уже исчерпаны, остается сделать только это. Однажды нам пришлось переносить большую программу форматирования текста из ее родной среды Unix на IBM PC. Компиляция программы прошла успешно, а вот при выполнении начали происходить очень странные вещи: пропадал примерно каждый второй символ из ее входных данных. Первая мысль, пришедшая нам в голову, была такова, что виноваты 16-разрядные целые числа, используемые вместо первоначальных 32-разрядных, или еще какая-нибудь проблема порядка байтов в переменных. Но путем вывода на экран символов, считываемых основным циклом программы, нам все-таки удалось обнаружить ошибку в стандартном заголовочном файле `ctype.h`, входящем в состав компилятора. В нем функция `isprint` определялась в виде функционального макроса:

```
?    #define isprint(c) ((c) >= 040 && (c) < 0177)
```

Главный рабочий цикл программы был основан на следующем условии:

```
?    while (isprint(c = getchar()))
?        ...
```

Всякий раз, когда поступающий на вход символ представлял собой пробел (его числовой восьмеричный код равен `040`; это заумный способ записи символа `' '`) или символ с более старшим кодом, а именно так и происходило в большинстве случаев, функция `getchar` вызывалась второй раз, поскольку в макрофункции аргумент вычислялся два раза. Таким образом, первый из введенных символов пропадал навсегда. Да, код программы был написан не так чисто, как следовало бы, — условие цикла содержало слишком много операций и допущений. Но заголовочный файл в данном случае содержал совершенно непростительную ошибку!

Время от времени можно встретить примеры аналогичных проблем; например, следующий макрос также определен в одном из заголовочных файлов другого компилятора:

```
?    #define __iscsym(c) (isalnum(c) || ((c) == '_'))
```

Существенной причиной проблемного поведения многих программ является "утечка" памяти — невозвращение ресурсов памяти, которые больше не используются. Также опасно не закрывать ненужные файлы, потому что со временем таблица открытых файлов переполняется, и программа больше не может открыть ни одного. Аварийное завершение программы с подобной утечкой выглядит особенно загадочно, потому что причиной его является недостаток некоторого ресурса, который не всегда удается воспроизвести.

Бывает, что и аппаратура компьютера дает сбои. В свое время в архитектуре процессора Pentium 1994 года выпуска была допущена ошибка в операциях с плавающей точкой; из-за этого многие вычисления давали неверные ответы. Эта ошибка широко обсуждалась и стоила больших денег, но как только ее удалось обнаружить, она, разумеется, стала легко воспроизводимой. Одна из самых странных ошибок, которую нам удавалось наблюдать, происходила давным-давно в программе-калькуляторе на двухпроцессорной машине. Иногда вычисление выражения `1/2` давало в итоге результат `0.5`, а иногда — нечто того же порядка, но совершенно неправильное, наподобие `0.7432`. При этом не было видно никакой закономерности в том, когда же программа дает правильный ответ, а когда — неправильный. В конце концов поиск ошибки привел нас к неисправному модулю работы с вещественными числами одного из процессоров. Поскольку программа-калькулятор выполнялась то одним, то другим процессором, ответы могли быть как правильными, так и бессмысленными без всякой закономерности.

Много лет назад мы работали на ЭВМ, внутреннюю температуру которой было легко измерить по количеству младших битов, вычисляемых неправильно при операциях над числами с плавающей точкой. Дело в том, что одна из микросхем неплотно сидела в своем гнезде; когда машина прогревалась, микросхема выходила из разъема чуть дальше и биты данных отсоединялись от соответствующих контактов.

5.5. Невоспроизводимые ошибки

Труднее всего работать с теми ошибками, которые “не стоят на месте”; как правило, при этом не удается свалить вину на сбои аппаратуры. Впрочем, сам факт недетерминированного поведения ошибки уже несет в себе наводящую информацию. Он означает, что ошибка, скорее всего, не имеет отношения к алгоритму; ее источник кроется в работе с теми данными, которые меняются всякий раз при очередном запуске программы.

Проверьте, все ли переменные инициализированы; ошибка может возникать оттого, что из неинициализированной ячейки памяти выбирается случайное значение, оставшееся там от предыдущих задач. Наиболее вероятные виновники таких ситуаций в C и C++ — это локальные переменные внутри функций и буферы памяти, выделенные функциями распределения памяти. Поэтому помещайте во все переменные известные начальные значения. Если в программе фигурирует инициализатор счетчика случайных чисел, который обычно формируется на основе текущего времени, принудительно сделайте его равным фиксированной константе — например, нулю.

Если ошибка изменяет свой характер или вообще исчезает при добавлении отладочного кода, то это, скорее всего, ошибка распределения памяти. В каком-то месте программы выполняется запись за пределами разрешенной области памяти, а добавление отладочных операторов изменяет структуру выделяемой памяти в достаточной степени, чтобы изменились и проявления ошибки. Большинство функций вывода, от `printf` до функций диалоговых окон, тоже участвуют в распределении памяти, еще больше мутя воду.

Если в месте аварийного завершения нет ничего похожего на ошибку, то проблема, скорее всего, связана с затиранием памяти: в определенный момент какое-то значение по ошибке помещается в ячейку памяти, которая будет использоваться гораздо позже. Иногда проблему создает “беспризорный” указатель на локальную переменную, который возвращается из функции при том, что переменная уничтожается, а потом по нему выполняется обращение. Вообще, возвращение адреса локальной переменной — верный путь к будущей катастрофе:

```
?     char *msg(int n, char *s)
?     {
?         char buf[100];
?
?         sprintf(buf, "error %d: %s\n", n, s);
?         return buf;
?     }
```

К тому времени, как произойдет обращение по указателю, возвращаемому из функции `msg`, он больше не будет указывать на место в памяти, содержащее какую-либо осмысленную информацию. Необходимо создавать буфер памяти с помощью функции `malloc`, использовать массив с модификатором `static` или же обеспечивать корректное распределение памяти в вызывающем модуле.

Аналогичные симптомы проявляются при попытке воспользоваться динамически выделенным буфером после того, как его освободили. Эта ситуация уже упоминалась в главе 2 в связи с функцией `freeall`. Вот этот код ошибочен:

```
?    for (p = listp; p != NULL; p = p->next)
?        free(p);
```

После освобождения памяти обращаться к ней больше не следует, поскольку ее содержимое может измениться, и нет никакой гарантии, что `p->next` по-прежнему будет указывать на нужный адрес памяти.

В некоторых реализациях функций `malloc` и `free` при двойном освобождении одного и того же буфера памяти повреждаются внутренние структуры данных. Однако проблема проявляется значительно позднее, когда программа наконец наталкивается на мешанину, которая возникла в результате такой операции. В некоторых функциях распределения памяти имеются отладочные режимы, после активизации которых корректность распределения проверяется при каждом вызове. Активизируйте такой режим, если у вас появилась ошибка недетерминистического характера. Если это не поможет, можно написать свою собственную функцию распределения, выполняющую часть необходимых проверок или ведущую протокол всех вызовов для последующего анализа. Такую функцию написать довольно легко, если нет необходимости заботиться о быстродействии. В запутанных ситуациях это помогает. Имеются превосходные коммерческие продукты, обеспечивающие управление памятью, проверку утечек и т.п.; если у вас нет к ним доступа, то хорошая альтернатива — это как раз написать свои собственные функции `malloc` и `free`.

Если программа работает у одного пользователя, но дает сбои у другого, то виновата может быть внешняя среда, в которой она выполняется. На работу программы влияют файлы, считываемые программой, допуски к файлам, системные переменные, заданные в системе стандартные пути, установки по умолчанию и т.п. Для таких ситуаций трудно посоветовать что-либо определенное. Обычно требуется разобраться в том, от каких параметров среды зависит программа, и скорректировать их соответственно.

Упражнение 5.1. Напишите версии функций `malloc` и `free`, которые можно было бы использовать при отладке программ с ошибками управления памятью. Один из подходов к реализации таких функций состоит в том, чтобы проверять все рабочее пространство при каждом вызове `malloc` или `free`, а другой — протоколировать всю информацию, которая может пригодиться для анализа с помощью других программ. В любом из случаев вставьте маркеры начала и конца в каждый выделяемый блок памяти для контроля переполнения с каждой из сторон.

5.6. Вспомогательные средства

Отладчик — это не единственное программное средство для поиска ошибок. Существует целый ряд программ, с помощью которых можно обработать объемистые выходные данные и найти в них важные фрагменты текста или аномальные явления, а также переупорядочить данные для более легкого их восприятия. Многие из таких программ входят в стандартные инструментарии разработчика, а некоторые написаны специально для поиска конкретных ошибок или анализа конкретной программы.

В этом разделе будет рассмотрена простая программа под названием `strings`, особенно полезная при чтении файлов, состоящих в основном из неотображаемых или непечатаемых символов, — исполняемых файлов и неразборчивых двоичных

потоков, используемых текстовыми редакторами для хранения форматированных текстов. В таких файлах часто содержится полезная информация — например, текст документа, сообщения об ошибках, недокументированные ключи или возможности, имена файлов и каталогов, имена вызываемых программой функций.

Мы обнаружили, что программа `strings` удобна для поиска текста и в других видах двоичных файлов. Файлы графических изображений часто содержат текстовые ASCII-строки, идентифицирующие те программы, с помощью которых данные изображения создавались. Сжатые файлы и архивы (например, в формате `zip`) могут содержать имена файлов. Программа `strings` может обнаружить и эту информацию.

В системах семейства Unix реализация программы `strings` уже есть, хотя и несколько отличается от приведенной в этом разделе. Она распознает исполняемые файлы (программы) и анализирует только их текстовые сегменты и разделы данных, игнорируя таблицы символов. Чтобы заставить ее просматривать весь файл, используется ключ `-a`.

Фактически программа `strings` извлекает ASCII-текст из двоичного файла для последующего чтения или обработки другими программами. Например, если сообщение об ошибке не несет никакой информации, можно не понять даже то, какая программа его выдала, не говоря уже о сути ошибки. В этом случае программу можно обнаружить, организовав поиск по вероятным каталогам с помощью команды такого типа:

```
% strings *.exe *.dll | grep 'непонятное сообщение'
```

Функция `strings` считывает файл и выводит все последовательности из не менее чем `MINLEN = 6` отображаемых символов.

```
/* strings: извлекает отображаемые символы из потока */
void strings(char *name, FILE *fin)
{
    int c, i;
    char buf[BUFSIZ];

    do {    /* один раз для каждой строки */
        for (i = 0; (c = getc(fin)) != EOF; ) {
            if (!isprint(c))
                break;
            buf[i++] = c;
            if (i >= BUFSIZ)
                break;
        }
        if (i >= MINLEN) /* вывести, если длина достаточна */
            printf("%s:%.*s\n", name, i, buf);
    } while (c != EOF);
}
```

Спецификация вывода `%.*s` в строке формата функции `printf` указывает взять длину строки из следующего аргумента (`i`), поскольку данная строка (`buf`) не заканчивается нулевым символом.

В цикле `do-while` каждая имеющаяся в файле строка последовательно обнаруживается и выводится на экран; его работа заканчивается, когда встречается конец файла (`EOF`). Проверка конца файла в самом конце цикла позволяет циклу с вызо-

вом getc и основному циклу функции завершиться по одному и тому же условию и при этом обойтись одним вызовом printf для работы с концом строки, концом файла и слишком длинной строкой.

Если бы использовался стандартный внешний цикл с проверкой в начале или один цикл с вызовом getc, но более сложным телом, то потребовалось бы дублировать вызов printf. Эта функция именно так и начинала свою жизнь, но в ней была ошибка, причем именно в операторе вызова printf. Мы исправили ее в одном месте, но забыли исправить еще в двух других. (Не забывайте задавать себе вопрос: "Не сделал ли я одну и ту же ошибку несколько раз?") В тот момент стало ясно, что лучше переписать программу и избежать дублирования кода; в результате возник цикл do-while.

Главная функция программы вызывает функцию strings для каждого из файлов, переданных в программу в качестве аргументов:

```
/* strings main: ищет отображаемые строки в файлах */
int main(int argc, char *argv[])
{
    int i;
    FILE *fin;

    setprogname("strings");
    if (argc == 1)
        eprintf("usage: strings filenames");
    else {
       for (i = 1; i < argc; i++) {
           if ((fin = fopen(argv[i], "rb")) == NULL)
               weprintf("can't open %s:", argv[i]);
           else {
               strings(argv[i], fin);
               fclose(fin);
           }
       }
    }
    return 0;
}
```

Должно быть, вы удивлены, что программа strings не считывает данные из стандартного потока ввода, если ей не заданы файлы-аргументы. В начале своего существования она это делала. Чтобы объяснить, почему сейчас это не так, необходимо рассказать историю ее отладки.

Самый очевидный тест программы strings состоял в том, чтобы дать ей для обработки ее же исполняемый файл. В системе Unix все работало нормально; в Windows 95 была задана следующая команда:

```
C:\> strings <strings.exe
```

Результат состоял ровно из пяти строк:

```
!This program cannot be run in DOS mode.
'.rdata
@.data
.idata
.reloc
```

Первая строка напоминает сообщение об ошибке, и мы потратили зря некоторое время, прежде чем поняли, что это текстовая строка из самой программы в скомпилированном виде. Не так уж редко случается, что сеанс отладки идет вкривь и вкось из-за неправильного понимания источника того или иного сообщения.

Итак, программа вроде бы выводит какие-то данные, но их явно должно быть больше. Где же они? Следующим вечером перед нами наконец забрезжил свет. ("Где-то я это уже видел!") Это была проблема переносимости, рассматриваемая более подробно в главе 8. Первоначально мы написали программу так, чтобы она считывала данные только из стандартного потока ввода с использованием функции `getchar`. Однако в системе Windows функция `getchar` возвращает `EOF` в том случае, если встречает определенный байт (`0x1A` или <Ctrl-Z>) во входном потоке. Это и было причиной преждевременного окончания работы программы.

Все это абсолютно корректно с точки зрения языка и системы, но явно не то, на что мы рассчитывали с нашим опытом работы в Unix. Мы собирались решить проблему, открыв файл для чтения в двоичном режиме (со спецификацией `"rb"`). Но поток `stdin` всегда открыт, и не существует стандартного способа изменить режим ввода из него. (Можно было бы воспользоваться функциями наподобие `fdopen` или `setmode`, но они не принадлежат к ANSI-стандарту С.) В конце концов мы встали перед необходимостью неприятного выбора одной из нескольких альтернатив: заставить пользователя всегда задавать имя файла (что корректно работает в Windows, но непривычно в Unix); молча выдавать неправильные ответы, если пользователь Windows пытается читать данные из стандартного потока ввода; использовать условную компиляцию для адаптации программы к разным средам, ухудшив тем самым переносимость программы. Мы выбрали первый вариант, чтобы одна и та же программа работала совершенно одинаково в разных средах.

Упражнение 5.2. Программа `strings` выводит строки, состоящие из `MINLEN` или большего количества отображаемых символов. Иногда это дает больше данных, чем хотелось бы. Усовершенствуйте программу так, чтобы в ней использовался необязательный аргумент, задающий минимальную длину строки.

Упражнение 5.3. Напишите программу `vis`, копирующую входной поток в выходной с заменой всех неотображаемых байтов (управляющих символов и т.п.), а также символов за пределами набора ASCII, последовательностями вида `\X`*hh*, где *hh* — шестнадцатеричное представление соответствующего байта. В противоположность программе `strings`, программа `vis` наиболее полезна для обработки потоков с совсем небольшим количеством неотображаемых символов.

Упражнение 5.4. Какой результат выдаст программа `vis`, если подать ей на вход поток вида `\X0A`? Как сделать результаты ее работы однозначными и недвусмысленными?

Упражнение 5.5. Усовершенствуйте программу `vis` так, чтобы она обрабатывала последовательности файлов, сворачивала длинные строки по любой заданной позиции, а также вообще удаляла неотображаемые символы. Какие еще возможности можно добавить в эту программу, исходя из ее назначения?

5.7. Ошибки, сделанные другими

Будем реалистами: большинство программистов не прельщает перспектива разработки совершенно новой системы с нуля. Вместо этого программисты в основном заняты переносом, доработкой, исправлением и (неизбежно) отладкой кода, написанного кем-то другим.

При отладке чужого кода сохраняют свою силу все принципы и рекомендации, которые мы привели в связи с отладкой своего собственного. Правда, перед отладкой такого кода вначале необходимо составить некоторое представление о том, как организована программа, как мыслили и писали ее авторы. Такая работа в отношении одного очень большого программного проекта была названа "разведкой", и это небезосновательная метафора. Задача, стоящая перед программистами, заключается в том, чтобы разведать, что и как происходит в системе, которую они впервые видят, так сказать, изнутри.

В этой работе очень существенную помощь могут оказать программные средства автоматизации. Все вхождения тех или иных имен в тексте можно найти с помощью программ текстуального поиска типа `grep`. Программы анализа перекрестных ссылок дают некоторое представление о структуре программы. Полезно иметь перед глазами схему графа функциональных вызовов в программе, если только она не слишком велика. Последовательность событий в программе восстанавливается путем пошагового выполнения в среде отладчика. История версий и изменений программы дает понятие об эволюции программы во времени. Частые изменения отдельных частей программы сигнализируют о том, что авторы плохо понимали работу данных фрагментов или непрерывно приспосабливали их к меняющимся требованиям. В обоих случаях такие части программы особенно подвержены ошибкам.

Иногда приходится искать ошибки в программах, разработка которых не в вашей власти, и даже их исходного кода у вас нет. В этом случае задача заключается в том, чтобы идентифицировать и охарактеризовать ошибку достаточно подробно, составив полный отчет о ее проявлениях, и в то же время найти способ избежать этой ошибки при пользовании программой.

Если вам кажется, что вы нашли ошибку в чьей-то программе, постарайтесь абсолютно точно убедиться, что ошибка действительно существует, иначе вы зря потратите время автора программы и подорвете доверие к себе.

Если на первый взгляд обнаруживается ошибка в компиляторе, необходимо сначала убедиться, что дело не в коде, а действительно в компиляторе. Например, в языках C и C++ не определено, должна ли операция поразрядного сдвига вправо заполнять очищенные биты нулями (логический сдвиг) или значением знакового бита (арифметический сдвиг). Поэтому новички часто обвиняют компилятор, если приведенная ниже конструкция дает неожиданный ответ.

```
?    i = -1;
?    printf("%d\n", i >> 1);
```

На самом же деле это проблема совместимости и переносимости, потому что такая операция выполняется по-разному на разных платформах. Поэтому следует протестировать код в разных средах, чтобы понять, что же в действительности происходит. Стоит также обратиться к описаниям и стандартам языка.

Убедитесь, что вы имеете дело с новой ошибкой. Имеется ли у вас новейшая версия программы? Прилагается ли к ней список исправленных ошибок? Большинство публично распространяемых программ за срок своего существования выпускается во множестве версий; если вы нашли ошибку в версии 4.0b1, она уже вполне может быть исправлена (или заменена на другую) в версии 4.04b2. Обычно программисты имеют привычку исправлять ошибки только в самых последних, текущих версиях своих программ.

Наконец, поставьте себя на место человека, для которого вы пишете отчет об ошибке. По возможности, владелец программы должен получить исчерпывающие тестовые данные, полностью характеризующие ошибку. Помощь с вашей стороны не покажется слишком эффективной, если ошибку можно будет продемонстрировать только на огромном массиве входных данных, или в сложной выполняющей среде, или в специальном файловом окружении. Сделайте тестовый набор данных минимальным и самодостаточным. Включите в отчет всю существенную информацию, такую как нужные версии самой программы, компилятора, операционной системы и аппаратных компонентов. Например, для ошибочной версии функции `isprint`, рассмотренной в разделе 5.4, можно предложить следующую тестовую программу:

```
/* тестовая программа для функции isprint */
int main(void)
{
    int c;

    while (isprint(c = getchar()) || c != EOF)
        printf("%c", c);
    return 0;
}
```

В качестве тестовых данных подойдет любая строка отображаемых символов, поскольку на выходе все равно будет получена всего половина входного потока:

```
% echo 1234567890 | isprint_test
24680
%
```

Лучший отчет об ошибке должен содержать тестовые исходные данные в количестве одной-двух строк; программа должна выполняться с этими данными в простейшей операционной среде и при этом гарантированно демонстрировать ошибку. Желательно, чтобы в совокупности вся эта информация позволяла легко решить проблему. Короче говоря, представляйте другим такие отчеты об ошибках, какие вы хотели бы получить сами.

5.8. Резюме

При правильном отношении к делу отладка может стать даже развлечением чем-то вроде решения головоломки. Но как бы ни относился к ней программист как к развлечению или как к тяжелой работе, — все равно это искусство, в котором приходится упражняться регулярно. Было бы хорошо, если бы ошибки вообще не случались; вот для чего мы стараемся писать программы в хорошем стиле с само-

го начала. Хорошо написанный код содержит мало ошибок, а те, которые в нем возникают, легче найти и исправить.

Как только обнаруживается ошибка, надо сразу же хорошенько задуматься над ее симптомами. Как она возникает? Знакомы ли вам ее проявления? Какие изменения вносились в программу в последнее время? Что такого особенного в исходных данных, которые ее спровоцировали? Для решения проблемы может хватить нескольких удачно подобранных тестовых примеров и пары операторов контрольного вывода в тексте программы.

Если симптомы неочевидны, следует задуматься еще сильнее. Хорошо подумав, нужно браться за локализацию ошибки, следуя при этом определенной системе. Первый шаг — это урезать входные данные до минимума, при котором ошибка все еще проявляется. Второй шаг — урезать код аналогичным образом, чтобы оставить в нем только аварийно-опасные фрагменты. Вполне можно вставить несколько контрольных операторов, активизируемых только на определенном этапе выполнения программы; это также помогает локализовать проблему. Все это примеры более общей стратегии — "разделяй и властвуй", — которая столь же эффективна в программировании, как в политике и на войне.

Прибегайте к вспомогательным средствам. Объясните ваш код кому-то другому (хотя бы и плюшевому медведю); это очень эффективно. Получите стековый фрейм с помощью отладчика и проанализируйте его. Воспользуйтесь коммерческими программами, способными проверять утечки памяти, нарушения границ массивов и другие подозрительные конструкции. Если станет ясно, что ваше представление о работе кода сильно расходится с действительностью, выполните программу в пошаговом режиме.

Постарайтесь лучше узнать себя и те ошибки, которые вы имеете привычку делать. Найдя и устранив ошибку, не пропустите остальные аналогичные недочеты в коде. Хорошенько обдумайте случившееся, чтобы по возможности не допустить его повторения.

Дополнительная литература

Много полезных рекомендаций по отладке кода можно найти в книгах Steve Maguire, *Writing Solid Code* (Microsoft Press, 1993) и Steve McConnell, *Code Complete* (Microsoft Press, 1993).

6

Тестирование

В практике вычислений, выполняемых вручную или с помощью настольных вычислительных машин, обычно тщательно проверяют процесс расчета, и если обнаруживается ошибка, то ее локализуют с помощью обратной процедуры, которая начинается с первого места ее проявления.

Норберт Винер. "Кибернетика"
(Norbert Wiener, Cybernetics*)*

Тестирование и отладку нередко объединяют в одну процедуру, но все же это разные операции с разными целями. Если упростить их суть до предела, то можно сказать, что отладка — это то, что делается, когда известно, что программа работает ошибочно и ее нужно исправить. Тестирование же — это систематические и настойчивые попытки сломить сопротивление нормально работающей программы и заставить ее сделать ошибку.

Эдсгер Дейкстра (Edsger Dijkstra) как-то сделал знаменитое наблюдение, что тестирование способно продемонстрировать наличие ошибок, но не их отсутствие. Он надеялся добиться такого уровня разработки программ, чтобы по самой своей конструкции они могли быть только правильными, не содержали ошибок и не требовали бы тестирования. Это прекрасная и достойная цель, но пока, к сожалению, недостижимая в программах сколько-нибудь существенного размера. Поэтому в этой главе мы сосредоточим внимание на том, как проводить тестирование с целью быстрого и эффективного поиска ошибок.

Хороший задел для успешного тестирования — это обдумывание потенциальных проблем по ходу написания кода. Систематическое тестирование, включающее в себя как самые простые, так и очень сложные, подробные тесты, помогает гарантировать правильность программы с самого начала и на всем протяжении ее существования. Автоматизация позволяет сократить объем ручной работы и поощряет программиста к тестированию. Наконец, существует множество особых приемов, которым каждый программист учится на собственном опыте.

Один из верных способов написания безошибочного кода — это его генерирование специальной программой. Если какая-нибудь задача программирования изучена и формализована настолько хорошо, что ее реализация является совершеннейшей рутиной, то ее желательно механизировать. Общий случай такой механизации — это генерирование программы на основе спецификации, написанной на некотором специализированном языке. Например, программы на языках высокого уровня транс-

лируются в код на языке ассемблера; для задания образцов поиска в тексте используются регулярные выражения; для определения операций над группами ячеек в электронных таблицах используются такие условные обозначения, как SUM(A1:A50) и т.д. В таких случаях, если генератор или транслятор работает правильно и спецификация также составлена правильно, то получившаяся в результате программа просто обязана быть правильной. Эта обширная тема будет рассмотрена более подробно в главе 9, а в этой главе мы вкратце поговорим о способах разработки тестов на основе компактных спецификаций.

6.1. Тестирование по мере написания кода

Чем раньше найдена проблема, тем лучше. Если систематически обдумывать то, что вы пишете, прямо по ходу написания, то простейшие свойства программы можно проверять непосредственно при ее создании, и в результате программа пройдет первую, тривиальную, стадию тестирования еще до компиляции. Некоторые виды ошибок можно таким образом вообще устранить из своей практики.

Проверяйте граничные условия и предельные случаи. Одним из эффективных методов является *проверка граничных условий*: по мере написания небольших блоков кода — например, циклов или условных операторов — проверяйте на месте, разветвляется ли выполнение по правильному пути и выполняется ли цикл нужное количество раз. Этот процесс называется проверкой граничных условий (или предельных случаев) по той причине, что контроль правильности выполняется на естественных границах кода и данных. К этой же технике относится проверка работы с несуществующим, пустым или состоящим из одного элемента потоком ввода, с целиком заполненным буфером и т.п. Главная идея состоит в том, что ошибки как раз чаще всего случаются на подобных границах. Если фрагмент кода может потерпеть крах, это скорее всего будет связано с одним из возможных предельных случаев. Соответственно, если фрагмент кода нормально работает на границах, он скорее всего будет корректно работать и в остальных ситуациях.

В следующем фрагменте, имитирующем работу функции fgets, из потока считываются символы, пока не встретится символ конца строки или пока не переполнится буфер:

```
?    int i;
?    char s[MAX];
?
?    for (i = 0; (s[i] = getchar()) != '\n' && i < MAX-1; ++i)
?        ;
?    s[--i] = '\0';
```

Представьте себе, что вы только что написали этот цикл. Прокрутите его мысленно, считывая воображаемую строку. Первое граничное условие, которое нужно проверить, имеет самый простой вид — это пустая строка. Возьмем текстовую строку, не содержащую ничего, кроме символа конца строки '\n'. Легко заметить, что цикл закончится на первом же проходе и переменная i останется равной 0. В последней строке кода эта переменная уменьшится на единицу и станет равной -1,

после чего в позицию s[-1], т.е. перед началом символьного буфера, будет записан нулевой байт. Проверка предельного случая сразу же обнаружила ошибку.

Если переписать код с применением стандартной конструкции для заполнения массива вводимыми из потока символами, то получится следующее:

```
?    for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n')
?            break;
?    s[i] = '\0';
```

Повторяя проверку, убеждаемся, что пустая строка теперь обрабатывается корректно: переменная i равна нулю, на первом же символе цикл заканчивает работу, и в позицию s[0] помещается символ конца строки '\0'. Проанализировав для верности работу цикла со строками длиной 1 или 2 символа, завершаемыми символами конца строки, можно убедиться, что в районе этого предельного случая код работает правильно.

Но есть и другие граничные условия, которые необходимо проверить. Если входной поток содержит слишком длинную строку или вообще не содержит символов конца строки, это проверяется сравнением i с верхним предельным значением MAX-1. Но что если входной поток пуст и первый же вызов функции getchar возвращает EOF? Это также необходимо контролировать:

```
?    for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?            break;
?    s[i] = '\0';
```

Проверка предельных случаев может выловить множество разных ошибок, но, к сожалению, не все. Мы вернемся к этому примеру в главе 8 и покажем, что в нем существует еще и проблема переносимости.

Следующий шаг — это проверить предельный случай на другой границе, когда массив практически заполнен, заполнен в точности до единого элемента, а также переполнен; в частности, надо проверить, что будет, если в момент переполнения из потока как раз поступает символ конца строки. Опустим подробности и оставим этот анализ читателю в качестве упражнения. При обдумывании предельных случаев возникает вопрос, что делать, когда буфер заполняется непосредственно перед поступлением символа '\n'; этот пробел в спецификации алгоритма необходимо заполнить как можно раньше, а обнаружить его помогает как раз проверка граничных условий.

Такие проверки помогают также обнаружить ошибки подсчета объектов, связанные с занижением или завышением их количества на единицу. По мере накопления опыта программист начинает выполнять эти проверки подсознательно, и многие тривиальные ошибки устраняются еще до того, как попадут в программу.

Проверяйте предусловия и постусловия. Еще один способ профилактики ошибок состоит в том, чтобы убедиться, что некоторые ожидаемые или необходимые условия выполняются до (предусловия) и после (постусловия) определенного фрагмента кода. Распространенный случай предусловий — это проверка того, что введенные исходные данные принадлежат к требуемому диапазону. Следующая

функция для вычисления среднего арифметического n элементов массива работает неправильно, если n меньше или равно нулю:

```
?   double avg(double a[], int n)
?   {
?       int i;
?       double sum;
?
?       sum = 0.0;
?       for (i = 0; i < n; i++)
?           sum += a[i];
?       return sum / n;
?   }
```

Что должна делать функция avg, если аргумент n равен нулю? Массив без элементов — это вполне реальное понятие, а вот его среднее значение — нет. Следует ли функции avg позволять системе перехватывать деление на 0? Аварийно завершаться? Сообщать об ошибке? Возвращать некое условное безобидное значение? Что если число n отрицательно (это лишено математического смысла, но вполне может случиться)? Как предлагалось в главе 4, предпочтительнее всего возвратить 0 в качестве среднего значения, если аргумент n меньше или равен нулю:

```
return n <= 0 ? 0.0 : sum/n;
```

И все же однозначно правильного ответа на возникшие вопросы мы так и не дали. А вот гарантированно неправильный ответ — это вообще игнорировать проблему. В одной статье из журнала *Scientific American* за ноябрь 1998 года рассказывается об инциденте, произошедшем на американском военном корабле "Yorktown" — крейсере, несущем управляемые ракеты. Один из членов команды непреднамеренно ввел 0 в ответ на запрос элемента данных. В результате произошло деление на нуль, эта ошибка вызвала каскад других ошибок, и в конечном итоге отключилось управление двигательной системой корабля. Крейсер безвольно дрейфовал в море в течение нескольких часов только потому, что управляющая программа не проверяла корректность входных данных.

Используйте контрольные условия. В языках C и C++ имеются средства для проверки контрольных предусловий и постусловий, объявленные в заголовочном файле <assert.h>. Если контрольное условие не выполняется, программа завершается аварийно, поэтому такие условия приберегают для ситуаций действительно неожиданных, в которых причина краха неизвестна, а дальнейшая работа программы невозможна. Приведенный выше код можно дополнить проверкой контрольного условия перед циклом:

```
assert(n > 0);
```

Если контрольное условие нарушается, программа завершается аварийно, выдавая стандартное сообщение:

```
Assertion failed: n > 0, file avgtest.c, line 7
Abort(crash)
```

Контрольные условия особенно полезны для проверки правильности работы интерфейсов, поскольку привлекают внимание к несогласованности между вызываю-

щими и вызываемыми модулями, и даже иногда помогают определить, кто из них виноват. Если контрольное условие того, что n больше нуля, не выполняется при вызове функции, то это однозначно указывает на источник проблемы в вызывающем модуле, а не в самой функции avg. Если интерфейс изменяется, а программист забывает внести соответствующие исправления в модуль, обращающийся к этому интерфейсу, то контрольное условие может помочь перехватить ошибку до того, как она действительно натворит бед.

Предусматривайте все случаи. Хорошей практикой следует считать добавление в программу кода на случай ситуаций, которые "не могут произойти". Здесь имеется в виду, что некоторые ошибки никак не могут случиться по вине программы, однако все же возникают в силу посторонних непредсказуемых обстоятельств. Один из примеров — это проверка отрицательной или нулевой длины массива в функции avg. Другой пример: в программе для обработки экзаменационных оценок вроде бы нет причин ожидать очень больших или отрицательных чисел во входном потоке, но проверять это все равно необходимо:

```
if (grade < 0 || grade > 100) /* невозможный случай */
    letter = '?';
else if (grade >= 90)
    letter = 'A';
else
    ....
```

Это пример безопасного стиля программирования, при котором программа сама защищает себя от неправильно введенных данных или другого некорректного обращения. Всегда можно предотвратить или корректно обработать такие ситуации, как нулевые значения указателей, выход за пределы массивов, деление на нуль и т.п. Если бы в управляющей программе на крейсере "Yorktown" были предусмотрены подобные меры, описанный выше инцидент с делением на нуль мог бы вовсе не произойти.

Проверяйте коды ошибок, возвращаемые из функций. Одна из защитных мер, которыми часто пренебрегают, — это проверка того, не возвратила ли библиотечная или системная функция код ошибки вместо кода нормального завершения. Значения, возвращаемые функциями ввода наподобие fread и fscanf, следует проверять обязательно, как и результаты операций открытия файлов с помощью функции fopen. Если открытие файла или чтение из него невозможно, то дальнейшие вычислительные операции не имеют смысла и не могут продолжаться корректно.

Проверка кода, возвращаемого функциями вывода наподобие fprintf или fwrite, позволяет обнаружить ошибку, возникающую из-за попытки записи на диск, где не хватает свободного пространства. Иногда бывает достаточно проверить значение, возвращаемое функцией fclose, которое равно EOF, если в процессе выполнения какой-либо операции с файлом случается ошибка, и 0 — в противном случае.

```
fp = fopen(outfile, "w");
while (...)                /* запись в outfile */
    fprintf(fp, ...);
if (fclose(fp) == EOF) {   /* есть ошибки? */
    /* обработка ошибки вывода */
}
```

Ошибки вывода могут иметь самые серьезные последствия. Если, например, записываемый файл является новой версией файла, содержащего ценную (а то и бесценную) информацию, то проверка ошибок позволит сохранить старый файл, если вдруг новый не записался корректно.

Трудозатраты при тестировании прямо по ходу написания программы минимальны и окупаются сторицей. Если сразу в ходе разработки думать о тестировании, то и код получается лучшего качества, потому что именно тогда-то вы и знаете лучше всего, что и как должна делать программа. Если же вместо этого ждать, пока не проявятся ошибки, то к тому времени можно забыть многие подробности работы кода. Придется вспоминать все заново, а это требует времени и усилий. В результате в условиях цейтнота не удастся выполнить настолько тщательную и всестороннюю коррекцию программы, насколько следовало бы, потому что ее знание и понимание непременно ухудшится за время, прошедшее с момента ее написания.

Упражнение 6.1. Выполните проверки граничных условий, предельных случаев в приведенных ниже программах, а затем внесите исправления в соответствии с принципами стилистики, изложенными в главе 1.

а) Вычисление факториала:

```
?    int factorial(int n)
?    {
?        int fac;
?        fac = 1;
?        while (n--);
?            fac *= n;
?        return fac;
?    }
```

б) Вывод символов текстового буфера по одному в каждой строке:

```
?    i = 0;
?    do {
?        putchar(s[i++]);
?        putchar('\n');
?    } while (s[i] != '\0');
```

в) Копирование строки текста из одного буфера в другой:

```
?    void strcpy(char *dest, char *src)
?    {
?        int i;
?
?        for (i = 0; src[i] != '\0'; i++)
?            dest[i] = src[i];
?    }
```

г) Еще одна процедура копирования, на этот раз `n` символов из строки `s` в `t`:

```
?    void strcnpy(char *t, char *s, int n)
?    {
?        while (n > 0 && *s != '\0') {
?            *t = *s;
?            t++;
```

```
?           s++;
?           n--;
?       }
?   }
```

д) Сравнение чисел:

```
?   if (i < j)
?       printf("%d is greater than %d.\n", i, j);
?   else
?       printf("%d is smaller than %d.\n", i, j);
```

е) Классификация символов:

```
?   if (c >= 'A' && c <= 'Z') {
?       if (c <= 'L')
?           cout << "first half of alphabet";
?       else
?           cout << "second half of alphabet";
?   }
```

Упражнение 6.2. Эта книга писалась в конце 1998 года, когда перед миром встала, пожалуй, самая грандиозная проблема "проверки предельного случая" — проблема 2000 года.

а) Какие даты вы бы использовали для тестирования работы системы в 2000 году? Предположим, что это тестирование стоит достаточно дорого. В каком порядке вы проверяли бы даты после самого очевидного случая — 1 января 2000 года?

б) Как бы вы протестировали стандартную функцию `ctime`, возвращающую строковое представление даты в следующей форме:

```
Fri Dec 31 23:58:27 EST 1999\n\0
```

Предположим, что функция `ctime` вызывается из вашей программы. Как бы вы защитили ваш код от возможной некорректной реализации этой функции?

в) Опишите процесс тестирования программы-календаря, выводящей результат в следующем виде:

```
      January 2000
 S   M Tu  W Th  F  S
                    1
 2   3  4  5  6  7  8
 9  10 11 12 13 14 15
16  17 18 19 20 21 22
23  24 25 26 27 28 29
30  31
```

г) Какие еще даты являются ключевыми в системах и программах, которыми вы пользуетесь, и как бы вы протестировали корректность работы с ними?

6.2. Систематическое тестирование

Очень важно тестировать программу по строгой системе, четко представляя себе на каждом шаге, что именно тестируется и какие результаты должны быть получены. Необходимо соблюдать порядок и последовательность, чтобы ничего не пропустить, и вести записи, чтобы не забыть, какая часть работы уже проделана.

Тестируйте маленькими шажками. Тестирование должно идти параллельно с выстраиванием структуры программы. Тестирование всей программы сразу, после ее написания, отнимает гораздо больше времени и усилий, чем пошаговое. Напишите часть программы, протестируйте ее, добавьте немного нового кода, опять протестируйте и т.д. Если у вас имеется два пакета модулей, которые пишутся и тестируются независимо, проведите их тестирование тогда, когда придет время соединить их в одно целое.

Например, когда мы тестировали программу CSV (см. главу 4), первый шаг состоял в том, чтобы написать ровно столько кода, сколько требуется для чтения входных данных. Это позволило проверить корректность ввода данных и их предварительной обработки. Следующим шагом было разбиение входных строк по запятым. Как только эти части заработали правильно, мы перешли к работе с полями в кавычках, а затем постепенно дошли до тестирования всей программы.

Начинайте тестирование с самых простых случаев. Пошаговый подход применим также и к выбору последовательности программных средств, которые следует протестировать. Вначале следует сосредоточить внимание на самых простых частях программы, составляющих ее основное, рутинно выполняемое тело. Только убедившись в правильности их работы, следует двигаться дальше. Таким образом, на каждом этапе тестированию подвергается все самое главное, и постепенно накапливается уверенность, что основные механизмы программы работают надежно. Простые ошибки обнаруживаются простыми тестами. Каждый тест вносит свой небольшой вклад в устранение следующей потенциальной проблемы. Хотя каждую следующую ошибку труднее заставить произойти, чем ее предшественницу, это не обязательно означает, что ее труднее исправить.

В этом разделе мы поговорим о выборе эффективных тестов и порядке их выполнения; в следующих двух — о механизации процедуры тестирования, чтобы ее можно было выполнять с высокой производительностью. Первым шагом, по крайней мере для небольших программ или отдельных функций, является обобщение проверки предельных случаев и граничных условий, которые рассматривались в предыдущем разделе, — систематическое тестирование простых случаев.

Предположим, есть функция, выполняющая двоичный поиск в массиве целых чисел. Следует начинать с таких тестов в порядке их усложнения:

- поиск в массиве, вообще лишенном элементов;
- поиск в массиве из одного элемента по ключу, который:
  * меньше, чем единственный элемент в массиве;
  * равен единственному элементу;
  * больше, чем единственный элемент;

- поиск в массиве из двух элементов по ключам, которые:
  * охватывают все пять возможных случаев;
- проверка корректности поиска в массиве с одинаковыми элементами по ключам, которые:
  * меньше, чем значение в массиве;
  * равны этому значению;
  * больше этого значения;
- поиск в массиве из трех элементов в таком же режиме, как из двух;
- поиск в массиве из четырех элементов в таком же режиме, как из двух и трех.

Если функция пройдет все эти тесты и не потерпит крах, то скорее всего она работает нормально, однако дальнейшее тестирование все же не исключается.

Описанный набор тестов достаточно невелик, чтобы его можно было выполнить от руки, но все-таки лучше написать небольшую программу автоматизированного тестирования, механизирующую этот процесс. Приведенная ниже программа настолько проста, насколько это вообще возможно. Она считывает строки, содержащие ключ поиска и размер массива, затем создает массив указанной длины из чисел 1, 3, 5,..., и разыскивает заданный ключ в массиве.

```
/* bintest main: автоматизированное тестирование binsearch */
int main(void)
{
    int i, key, nelem, arr[1000];

    while (scanf("%d %d", &key, &nelem) != EOF) {
        for (i = 0; i < nelem; i++)
            arr[i] = 2*i + 1;
        printf("%d\n", binsearch(key, arr, nelem));
    }
    return 0;
}
```

Это довольно примитивная конструкция, но она демонстрирует, что полезная программа автоматизации тестирования не обязательно должна быть большой. Ее легко расширить и доработать так, чтобы она выполняла дополнительные тесты и требовала как можно меньшего объема ручной работы.

Знайте, чего ожидать на выходе. При тестировании всегда необходимо знать, какой правильный ответ должен получиться на выходе программы; если он неизвестен, тестирование становится пустой тратой времени. На первый взгляд здесь все очевидно. И действительно, по выходным данным многих программ можно сразу определить, работает программа или нет — например, является ли копия файла действительно точным его дубликатом или нет, выдает ли программа сортировки действительно отсортированный массив, и является ли он результатом перестановки исходного.

Но большинство программ не поддается такому простейшему анализу. Например, как определить, выдает ли компилятор точный перевод исходного кода на язык ассемблера или машинных кодов? Дает ли численный алгоритм ответ в заданных границах точности? Помещает ли графическое приложение все пиксели на поло-

женные им места? В таких программах особенно важно найти способ сравнения результата с известным правильным ответом.

- Для тестирования компилятора следует скомпилировать и запустить на выполнение тестовые файлы исходного кода. Эти тестовые программы должны выдавать выходные данные, которые можно будет сравнить с известными ответами.
- Для тестирования программы численного расчета постройте набор тестовых задач, охватывающих его предельные случаи, — как простейшие, так и более сложные. Везде, где это возможно, напишите код для анализа свойств выдаваемых ответов. Например, при тестировании программы численного интегрирования ответ следует проверять на непрерывность и на совпадение с известными аналитическими решениями в замкнутой форме.
- Чтобы протестировать программу для работы с графикой, недостаточно нарисовать на экране прямоугольник; считайте данные этого прямоугольника с экрана и убедитесь, что его контур в точности удовлетворяет заданным условиям.

Если программа преобразует данные в двух взаимно обратных направлениях, проверьте, может ли она в точности восстановить исходную информацию. Пример таких двух направлений — это шифровка и дешифровка данных. Если зашифрованные данные не удается расшифровать в их исходную форму, значит, что-то не так. Аналогично, взаимно обратными алгоритмами являются методы упаковки (сжатия) файлов и их распаковки. Если программа может упаковать набор файлов, она должна быть в состоянии распаковать их без потерь и искажений. Иногда обратную операцию можно проделать несколькими способами; в таком случае протестировать нужно все эти способы.

Проверяйте свойства сохранения. Многие программы при обработке данных сохраняют неизменными некоторые их свойства, которые бывает полезно проверить. Например, утилиты наподобие `wc` (для подсчета строк, слов и символов) или `sum` (для подсчета контрольной суммы) позволяют убедиться, что те или иные данные имеют одинаковый размер, одинаковое количество слов, содержат одни и те же байты в том же порядке и т.п. Существуют программы, которые устанавливают идентичность файлов (`cmp`) или, наоборот, генерируют список различий между ними (`diff`). Эти программы или их аналоги имеются во многих операционных средах, и их стоит иметь под рукой.

Программа подсчета частоты байтов, приведенная далее, может использоваться для проверки сохранности данных после обработки, а также для выявления аномалий наподобие нетекстовых символов в текстовых файлах. Вот одна из версий этой программы, которую мы назвали `freq`:

```
#include <stdio.h>
#include <ctype.h>
#include <limits.h>

unsigned long count[UCHAR_MAX+1];

/* freq main: подсчет частоты байтов */
int main(void)
```

```
{
    int c;

    while ((c = getchar()) != EOF)
        count[c]++;

    for (c = 0; c <= UCHAR_MAX; c++)
        if (count[c] != 0)
            printf("%.2x  %c  %lu\n",
                c, isprint(c) ? c : '-', count[c]);
    return 0;
}
```

Внутри программы тоже можно проверять некоторые свойства сохранения. Тривиальная проверка самосогласованности состоит в том, чтобы подсчитать количество элементов в структуре данных с помощью специальной функции. Например, хэш-таблица должна быть устроена таким образом, чтобы каждый помещенный в нее элемент можно было впоследствии извлечь. Это свойство можно проверить, введя в программу функцию, выводящую все содержимое таблицы в файл или массив. В любой момент времени количество вставок в таблицу минус количество удалений из нее должно равняться количеству оставшихся элементов, и это условие достаточно легко проверить.

Сравнивайте независимые реализации. Независимые друг от друга реализации одной и той же библиотеки или программы, очевидно, должны давать одинаковые результаты. Например, два компилятора должны транслировать одну и ту же программу в исполняемые модули, которые бы вели себя одинаково в одной и той же выполняющей среде, по крайней мере, в большинстве ситуаций.

Иногда ответ можно вычислить двумя различными способами или же написать очень простую (до тривиальности) версию программы, которая делает то же самое, только медленно и нерационально; такая версия используется для независимого сравнения. Если две независимо разработанные программы дают один и тот же ответ, то они, скорее всего, работают правильно. Если же ответы различаются, то по крайней мере в одной из программ есть ошибка.

Один из авторов как-то раз разрабатывал компилятор для новой среды вместе с одним из сотрудников. Работа по отладке кода, генерируемого компилятором, была распределена: один человек писал программу, которая кодировала инструкции для выполняющей среды, а второй писал дизассемблер для отладчика. Вследствие этого было маловероятно дублирование любой ошибки в интерпретации или реализации набора инструкций сразу в двух компонентах. Если компилятор неправильно кодировал ту или иную инструкцию, дизассемблер обязательно это замечал. Все данные, получаемые на выходе компилятора, прогонялись через дизассемблер и сравнивались с отладочными данными, выдаваемыми самим компилятором. Этот подход оказался очень практичным и результативным; он позволил обнаружить и исправить много ошибок в обоих компонентах. Необходимость долгой и трудной отладки возникала лишь тогда, когда разработчики обоих компонентов одинаково неправильно интерпретировали двусмысленную фразу в спецификации архитектуры выполняющей среды.

Следите за полным охватом программы при тестировании. Одна из задач тестирования заключается в том, чтобы в процессе прогона тестов заставить выполниться каждый оператор программы. Тестирование не может считаться полным, если каждая строка программы не была выполнена хотя бы один раз. Подобного полного охвата при тестировании бывает не так-то легко добиться. Даже оставляя в стороне непредсказуемые ситуации, которые в нормальных условиях не возникают, бывает непросто заставить программу выполнить определенные операторы, задавая ей те или иные исходные данные.

На рынке программного обеспечения имеются готовые программы для измерения частоты выполнения отдельных частей программы. Это программы профилирования, которые часто входят в состав пакетов компиляторов или сред разработки и предоставляют возможность подсчитать частоту выполнения любых операторов программы. Это дает возможность определить, насколько полный анализ выполняемости программы дает тот или иной тест.

Мы протестировали программу из главы 3 для генерирования текста по алгоритму марковских цепей, используя комбинацию всех перечисленных методов. Эти тесты подробно описаны в последнем разделе данной главы.

Упражнение 6.3. Предложите процедуру тестирования для программы `freq`.

Упражнение 6.4. Спроектируйте и реализуйте версию программы `freq`, подсчитывающую частоты употребления других, не символьных, данных, — таких как 32-разрядные целые или вещественные числа. Можно ли сделать так, чтобы одна и та же программа просто и удобно выполняла одну и ту же операцию над разными данными?

6.3. Автоматизация тестирования

Выполнять большой объем тестирования вручную, во-первых, скучно, во-вторых, ненадежно. При надлежащем тестировании нужно выполнить очень много тестов, задать очень много исходных данных и сравнить очень много результатов. Поэтому тестирование желательно переложить на программы, которые никогда не устают и не проявляют небрежности. Стоит потратить некоторое время и написать сценарий на командном языке или тривиальную программу, которая выполняет все необходимые тесты. После этого полный набор тестовых примеров можно будет запускать нажатием одной кнопки (в буквальном или переносном смысле). Чем легче запускать набор тестов на выполнение, тем чаще это можно делать и тем меньше вероятность пропустить что-нибудь, если время будет поджимать. Мы написали тестовую программу, тестирующую весь код, написанный для этой книги, и запускали ее всякий раз, когда в код вносились какие-либо изменения. Отдельные части этого теста запускались автоматически после каждой успешной компиляции.

Автоматическое регрессионное тестирование. Начинать автоматизацию следует с *регрессионного тестирования* — последовательности тестов, сравнивающих новую версию какой-либо программы с ее же предыдущей версией. При исправлении ошибок и устранении проблем программисты имеют тенденцию тестировать только исправленные фрагменты кода, упуская из виду возможность того, что это же

самое исправление испортило что-то в другой части программы. Назначение регрессионного тестирования состоит как раз в том, чтобы гарантировать неизменную работу всех функций программы, кроме добавленных.

В некоторых системах средства для такой автоматизации имеются в изобилии. Так, языки разработки сценариев позволяют легко написать короткую тестовую процедуру для прогона набора тестов. В системе Unix имеются утилиты сравнения файлов, `diff` и `cmp`, позволяющие анализировать выходные данные программ на совпадение с эталонными. Утилита `sort` позволяет собирать родственные элементы данных вместе; утилита `grep` помогает отфильтровать выходные данные, а `wc`, `sum` и `freq` — резюмировать их. Все эти средства в совокупности позволяют быстро и эффективно разработать автоматизированные тестовые процедуры почти для любых программ, разработкой которых занимается один человек или небольшая группа. Правда, для тестирования больших программных проектов этих средств может оказаться недостаточно.

Далее приведен сценарий регрессионного тестирования для программы под названием `ka`. Этот сценарий запускает старую версию (`old_ka`), а затем новую (`new_ka`) на большом количестве разных тестовых файлов и сообщает, если между выходными данными двух версий имеются различия. Он написан на языке командной оболочки Unix, но его можно легко перевести на Perl или другой язык разработки командных сценариев.

```
for i in ka_data.*          # цикл по тестовым файлам данных
do
    old_ka $i >out1         # запуск старой версии
    new_ka $i >out2         # запуск новой версии
    if ! cmp -s out1 out2   # сравнение выходных данных
    then
        echo $i: BAD        # если разные, то ошибка
    fi
done
```

Сценарий тестирования обычно должен выполняться "молча", выводя какую-то информацию только в том случае, если происходит что-то неожиданное. Именно так работает приведенный сценарий. Вместо этого можно было бы выводить имя каждого обрабатываемого файла, а после него — сообщение об ошибке, если таковая случается. Подобная индикация процесса помогает выявить, например, такую проблему, как зацикливание или запуск не того теста в ходе выполнения сценария, но если тест выполняется правильно, излишняя информация на экране может раздражать *пользователя*.

Аргумент `-s` утилиты `cmp` приказывает ей возвращать код сравнения, но не выводить никаких данных на экран. Если сравнение файлов показывает их совпадение, то возвращается код "истина", выражение `! cmp` ложно и на экран ничего не выводится. Если же выходные данные старой и новой версий программы отличаются, то утилита `cmp` возвращает код "ложь" и выводятся имя файла и предупреждающее сообщение.

При выполнении регрессионного тестирования делается неявное предположение о том, что старая версия программы вычисляет правильный ответ. То, что это действительно так, следует очень скрупулезно проверить в самом начале работы над программой и затем тщательно поддерживать такое положение дел. Если в регрессион-

ный тест закрадется неправильный ответ, обнаружить его будет не так-то просто, а вся информация, базирующаяся на нем, окажется искаженной. Полезно время от времени выполнять тщательную проверку самого регрессионного теста, чтобы лишний раз убедиться в его правильности.

Самодостаточные тесты. В дополнение к регрессионному тестированию следует составить набор самодостаточных тестов, в состав которых входят исходные данные и эталонные результаты. Поучительным примером может служить наш опыт по тестированию интерпретатора Awk. Многие конструкции языка Awk тестировались таким образом: определенные данные прогонялись через маленькие программки на этом языке, а затем результаты сравнивались с эталонными. Ниже приведен фрагмент большого набора тестов, предназначенный для проверки работы одного хитроумного выражения инкрементирования. В этом тесте новая версия интерпретатора Awk (`newawk`) выполняется над короткой программой. Результат работы новой версии записывается в один файл, эталонный результат — в другой; затем файлы сравниваются, и выдается сообщение об ошибке, если они не совпадают.

```
# тест инкрементирования поля: $i++ означает ($i)++, а не $(i++)
echo 3 5 | newawk '{i = 1; print $i++; print $1, i}' >out1
echo '3 4 1' >out2          # правильный ответ
if ! cmp -s out1 out2       # различные данные на выходе
then
    echo 'BAD: field increment test failed'
fi
```

Первый комментарий является частью теста. В нем сообщается и документируется, какие именно возможности тестируются.

Иногда удается построить большое количество тестовых примеров сравнительно небольшими усилиями. Для целей анализа простых выражений мы создали простой узкоспециализированный язык описания тестов, исходных данных и ожидаемых результатов. Ниже приводится короткое описание небольшого теста, который проверяет корректность некоторых способов представления числа 1 в языке Awk.

```
try (if ($1 == 1) print "yes"; else print "no"}
1       yes
1.0     yes
1E0     yes
0.1E1   yes
10E-1   yes
01      yes
+1      yes
10E-2   no
10      no
```

Первая строка представляет собой тестируемую программу (все, что идет после слова `try`). В каждой последующей строке приведены исходные данные и ожидаемые результаты, разделенные символами табуляции. В первом тесте говорится, что если первое поле исходных данных равно 1, то на выходе программа должна сообщить `yes`. Первые семь тестов дают результат `yes`, а последние два — `no`.

Программа на языке Awk (а как же иначе!) преобразует каждый тест в полную программу на языке Awk, затем запускает ее на выполнение с каждым набором

входных данных и сравнивает результаты с эталонными. Какая-либо информация выводится на экран только тогда, когда выдается неправильный ответ, не совпадающий с эталонным.

Аналогичные механизмы используются для тестирования команд поиска по регулярным выражениям и замены/подстановки. Специальный язык для написания тестов (пусть даже примитивный) позволяет с удобством сгенерировать множество тестовых примеров. Создание программы для написания другой программы с целью тестирования третьей программы — это своеобразный рычажный механизм, позволяющий серьезно увеличить первоначально приложенное усилие. (В главе 9 о таких языках и программах, которые пишут другие программы, рассказывается более подробно.)

В целом мы разработали около тысячи тестов языка Awk, причем весь этот набор можно было запустить одной командой. Если все шло хорошо, на экран вообще ничего не выводилось. Как только в язык добавлялась новая возможность или в нем исправлялась ошибка, тут же разрабатывались новые тесты, позволявшие проверить их корректную работу. При каждом изменении программы, даже тривиальном, запускался весь набор тестовых примеров; его выполнение отнимало всего несколько минут. В результате иногда удавалось “поймать” совершенно неожиданные ошибки, а в целом этот набор тестов неоднократно спасал авторов языка Awk от публичного позора.

Что же делать, когда оказывается, что в программе есть ошибка? Если ее не обнаруживает ни один из имеющихся тестов, следует разработать новый тест и проверить его на ошибочной версии кода. Обнаруженная ошибка может дать идеи для дальнейших тестов или совершенно новых классов проверок. Иногда бывает достаточно добавить в программу средства защиты, перехватывающие и обрабатывающие ошибку внутри нее.

Сохраняйте все имеющиеся у вас тесты. С их помощью можно определить, описывает ли отчет об ошибке в очередной версии программы действительную ситуацию или она уже давно исправлена. Ведите записи об ошибках, изменениях и исправлениях; это поможет вспомнить старые проблемы и исправить новые. В большинстве организаций по разработке программного обеспечения это обязательный элемент рабочего процесса. Но если даже вы программируете для себя, такие записи все равно стоит делать, потому что времени это отнимает немного, а положительный результат налицо.

Упражнение 6.5. Разработайте набор тестовых примеров для тестирования функции `printf`, применяя как можно больше средств автоматизации.

6.4. Тестирование компонентов в программных оболочках

До сих пор наше изложение было посвящено тестированию законченных автономных программ. Но это не единственный способ автоматизированного тестирования, как и не самый удобный метод тестирования частей большой программы в процессе ее разработки, особенно если такая разработка выполняется на коллективной основе. Неудобен этот способ и тогда, когда необходимо протестировать небольшой компонент, встроенный в большую систему.

Чтобы протестировать отдельный компонент, для него обычно приходится писать некую программную оболочку, содержащую достаточно средств интерфейса, чтобы компонент мог выполняться корректно. Ранее в этой главе уже демонстрировалась самая примитивная оболочка такого вида для тестирования функции двоичного поиска.

Достаточно легко написать нечто подобное и для тестирования таких компонентов, как математические функции, строковые функции, процедуры сортировки и т.п., поскольку все, что необходимо сделать в оболочке, — это задать входные параметры, вызвать функцию и сравнить результат с эталоном. А вот чтобы протестировать таким образом частично незавершенную программу, требуется больше усилий.

Для иллюстрации сказанного пройдем все этапы построения программной оболочки для тестирования функции `memset`, одной из функций семейства `mem...` стандартной библиотеки C/C++. Эти функции часто пишутся на языке ассемблера для конкретной выполняющей среды, поскольку большое значение имеет их быстродействие. Чем больше их оптимизируют при разработке, тем больше вероятность ошибки и тем тщательнее необходимо их тестировать.

Первый шаг — это написать самую простую, но гарантированно работающую версию на языке C. Она даст возможность сравнения по быстродействию и, что еще важнее, образец правильной работы. При переходе в новую выполняющую среду переносится именно простейшая версия, а затем уже выполняются все оптимизации, пока не будет достигнуто нужное качество.

Функция `memset(s,c,n)` делает `n` байт памяти равными заданному байту `c`, начиная с адреса `s`, после чего возвращает `s`. Если не заботиться о быстродействии, то написать такую функцию проще простого:

```
/* memset: делает n байт по адресу s равными c */
void *memset(void *s, int c, size_t n)
{
    size_t i;
    char *p;

    p = (char *) s;
    for (i = 0; i < n; i++)
        p[i] = c;
    return s;
}
```

Но если соображения быстродействия играют решающую роль, то используются такие приемы, как запись целых 32- или 64-разрядных слов. При оптимизации таких операций часто возникают ошибки, поэтому необходимо самое тщательное тестирование.

Тестирование основывается на сочетании проверки предельных случаев и перебора вариантов в потенциально опасных местах программы. Для функции `memset` к предельным случаям относятся значения `n`, равные, во-первых, тривиальным числам нуль, один или два, во-вторых, степеням двойки и близлежащим числам. Это могут быть как маленькие, так и довольно большие числа, — например, 2^{16}, соответствующее естественному пределу во многих выполняющих средах с 16-разрядным словом. Степени двойки заслуживают внимания по той причине, что один из способов ускорить работу функции `memset` — это записывать сразу по несколько байт за один раз.

Для этого используются специальные машинные инструкции или запись целых слов вместо отдельных байтов. Аналогично, необходимо проверить различные способы размещения начала массива в памяти с различным выравниванием; это позволяет обнаружить ошибки, связанные с начальным адресом или длиной. Мы поместим основной массив в более длинный, таким образом создавая буферную зону или полосу безопасности с каждой его стороны, чтобы иметь возможность легко варьировать выравнивание.

Также необходимо проверить целый ряд значений аргумента `c`, в том числе нуль, `0x7F` (самое большое значение со знаком в 8-разрядных байтах), `0x80` и `0xFF` (провокации на ошибку обработки символов со знаком и без знака), а также несколько чисел, значительно больших, чем помещаются в один байт (чтобы убедиться, что действительно используется только один байт). Следует также инициализировать буфер памяти некоторым известным эталонным значением, отличающимся от всех перечисленных, чтобы проверить, записывает ли функция `memset` данные за пределами заданного массива.

Простейшую реализацию функции можно использовать как стандарт для сравнения в таком тесте, в котором в памяти размещаются два массива, а затем сравнивается работа функции с различными комбинациями `n` и `c` при нескольких возможных смещениях внутри массива:

```
big = макс. левое поле + макс. n + макс. правое поле
s0 = malloc(big)
s1 = malloc(big)
для каждой комбинации n, c и смещения offset:
    записать в s0 и s1 известный образец
    выполнить медленную memset(s0 + offset, c, n)
    выполнить быструю memset(s0 + offset, c, n)
    проверить возвращаемые значения
    сравнить s0 и s1 побайтово
```

Если в результате ошибки функция `memset` запишет что-нибудь за пределами заданного массива, то это скорее всего случится в начале или в конце буфера, так что наличие буферной зоны позволяет легко отследить поврежденные байты, а также снизить вероятность повреждения других частей программы. Чтобы выявить запись за пределами заданного буфера, мы сравниваем *все* байты массивов `s0` и `s1`, а не только те `n` байт, которые функция должна записывать.

Итак, разумно достаточный набор тестов должен включать проверку всех комбинаций следующих значений:

```
offset = 10, 11, ..., 20
c = 0, 1, 0x7F, 0x80, 0xFF, 0x11223344
n = 0, 1, 2, 3, 4, 5, 7, 8, 9, 15, 16, 17,
    31, 32, 33, ..., 65535, 65536, 65537
```

Значения `n` должны охватывать как минимум $2^i - 1$, 2^i и $2^i + 1$ для i от 0 до 16. Все эти числа не стоит перечислять в теле тестовой программной оболочки, а вместо этого включить в массивы, создаваемые от руки или автоматически. Лучше сгенерировать их автоматически; в этом случае легче задать больше степеней двойки, вариантов смещений и символов.

Эти тесты позволят полностью проанализировать работу memset, при этом не отнимая много времени даже на их создание, не говоря уже о выполнении. Всего описанный тестовый набор содержит около 3500 тестов. Все они полностью переносимы в другие операционные среды.

В качестве предупреждения расскажем следующую историю. Как-то раз мы передали копию тестовой программы и набора для memset разработчику одной операционной системы и библиотек для нового процессора. Несколько месяцев спустя мы (авторы исходного теста) начали пользоваться этой системой и вдруг обнаружили, что большая программа терпит крах при прогоне ее набора тестов. Мы проследили источник проблемы до крохотной ошибки, связанной со знаком в реализации функции memset на языке ассемблера. По неизвестной причине разработчик библиотеки изменил процедуру тестирования memset так, что она перестала проверять значения c больше 0x7F. Разумеется, ошибка была локализована путем выполнения исходного, правильного набора тестов, как только мы поняли, что главным подозреваемым является функция memset.

Такие функции, как memset, поддаются исчерпывающему тестированию методом перебора, поскольку достаточно просты, и можно доказать, что тестовые данные обеспечивают перебор всех возможных путей выполнения. Например, функцию memmove можно протестировать на все возможные комбинации наложения, перемещения и выравнивания буферов памяти. Это не есть перебор в точном смысле этого слова, поскольку не перебираются все возможные операции копирования, однако в тестовом наборе представлены все существенно различные ситуации.

Разумеется, как и в других случаях, программная оболочка должна содержать правильные ответы, с которыми сравниваются выходные данные тестируемого модуля. Важный метод, использованный нами при тестировании функции memset, заключается в том, чтобы сравнивать работу новой оптимизированной версии (возможно, ошибочной) с работой простейшей функции, в правильности которой мы уверены. Это можно делать в несколько этапов, как показывает описанный ниже пример.

Один из авторов занимался разработкой библиотеки для работы с растровой графикой. В ней применялась операция копирования блоков пикселей из одного изображения в другое. В зависимости от заданных параметров это могло быть простое копирование буферов памяти или же копирование с преобразованием из одной цветовой кодировки в другую; возможно было также "мозаичное" копирование, при котором фрагмент многократно дублировался в прямоугольной области, а также любые комбинации этой техники и нескольких других. Спецификация этой операции выглядела просто, а вот для эффективной ее реализации требовалось написать много специализированного кода для самых разных случаев. Проверка правильности работы всего этого кода потребовала специальной стратегии тестирования.

Вначале вручную была написана небольшая простая процедура, выполняющая данную операцию правильно над одним пикселем. Она использовалась для проверки того, как библиотека справляется с обработкой одного пикселя. После завершения этого этапа можно было быть уверенным, что с одним пикселем дела обстоят нормально.

Затем была написана процедура для работы с одной горизонтальной строкой пикселей. Она вызывала библиотечные функции для обработки одного пикселя за

раз, поэтому работала очень медленно. Результаты ее работы сравнивались с результатами, выдаваемыми гораздо более быстродействующей библиотечной процедурой. Когда все прошло успешно, библиотеке можно было уже доверять и в работе с горизонтальными строками пикселей.

Мы продолжали в том же духе: составили из строк прямоугольники, из прямоугольников — мозаику и т.д. По ходу дела было найдено множество ошибок, в том числе некоторые из них — в самой тестирующей программе. В этом и состояла сила метода: проверялись сразу два независимых компонента, и постепенно возрастало доверие к обоим. Если тест выполнялся неправильно, тестирующая программа выдавала подробный отчет обо всех обработанных данных; с его помощью можно было проанализировать правильность работы как библиотеки, так и самой тестирующей процедуры.

Данная библиотека дорабатывалась и переносилась в другие среды еще много лет, и всякий раз тестирующая программа помогала находить и исправлять новые ошибки. Поскольку в ней применялся послойный, многоуровневый подход, ее необходимо было каждый раз выполнять целиком, чтобы заодно проверять ее соответствие библиотеке. Так получилось, что выполняемое ею тестирование было не сплошным, а вероятностным: в ходе теста генерировались случайные тестовые данные, и они, накапливаясь за достаточно долгое время, позволяли протестировать буквально каждый оператор кода. Поскольку потенциальных тестовых данных могло быть очень много, эта стратегия оказалась намного эффективнее, чем построение тестового набора от руки или исчерпывающий перебор всех возможных тестовых примеров.

Упражнение 6.6. Разработайте тестирующую оболочку для функции `memset` согласно принципам, которые мы рассмотрели.

Упражнение 6.7. Разработайте тесты для всех остальных функций семейства `mem...`.

Упражнение 6.8. Выберите режим тестирования для таких математических функций С, как `sqrt`, `sin` и т.п., объявленных в заголовочном файле `math.h`. Какие входные данные имеет смысл задавать для их тестирования? Какие независимые проверки можно (и нужно) выполнить?

Упражнение 6.9. Разработайте механизмы для тестирования функций семейства `str...` языка С, например `strcmp`. Некоторые из этих функций, особенно функции выделения лексем `strtok` или `strcspn`, существенно сложнее, чем семейство `mem...`; соответственно, они требуют более сложного и тщательного тестирования.

6.5. Стрессовое тестирование

Один из эффективных методов тестирования состоит в том, чтобы “нагрузить” программу большим объемом исходных данных, сгенерированных автоматически. Такие данные вынуждают программу вести себя по-другому, чем в случае данных небольшого объема, введенных вручную. Большой объем данных сам по себе опасен, поскольку из-за него переполняются буферы, массивы, счетчики и т.п.; это позволяет обнаружить в программе места, где размеры структур данных сделаны фиксирован-

ными или не контролируются должным образом. Пользователь-человек имеет тенденцию избегать "невозможных" случаев наподобие пустых потоков ввода, числовых данных, выходящих за пределы заданного диапазона, слишком длинных имен или несообразно больших значений параметров. А вот компьютерные программы генерируют исходные данные в строгом соответствии с заданным алгоритмом, без всяких попыток уклониться от той или иной конкретной ситуации.

Для иллюстрации ниже приведена одна строка данных, сгенерированных компилятором Microsoft Visual C++ версии 5.0 при трансляции STL-варианта программы `markov`. Строка отредактирована так, чтобы помещаться на страницу.

```
xtree(114) : warning C4786: 'std::_Tree<std::deque<std::
basic_string<char,std::char_traits<char>,std::allocator
<char>>,std::allocator<std::basic_string<char,std::
 ... опущено 1420 символов
allocator<char>>>>>>::iterator' : identifier was
truncated to '255' characters in the debug information
```

Компилятор предупреждает, что он сгенерировал имя переменной, имеющее невероятную длину 1594 символа, но в отладочной информации сохранил только его первые 255 символов. Далеко не все программы способны обезопасить себя от негативного влияния строк подобной длины.

Задание случайных наборов входных данных (не обязательно корректных) — это еще один способ агрессивной атаки на программу в надежде сломить ее сопротивление. Это делается вместо того, чтобы рассуждать на тему "нормальные люди так не поступают". Например, некоторые коммерческие компиляторы С тестировались с помощью сгенерированных случайным образом, но синтаксически правильных исходных текстов. Прием здесь состоит в том, чтобы взять спецификацию данных — в данном случае стандарт языка С — и написать программу, генерирующую данные в соответствии с этой спецификацией, но по случайной схеме.

Такие тесты рассчитаны скорее на проверку встроенных в программу средств контроля и самозащиты, а не на сравнение полученных ответов с эталонными; их целью является спровоцировать крах программы или "невозможную" ситуацию, а не обнаружить ошибки в алгоритме. Они же являются хорошим средством для проверки встроенных механизмов обработки ошибок. Если исходные данные имеют разумный характер, большая часть ожидаемых ошибок выполнения не случается и код для их обработки бездействует; однако именно в таких углах в основном и прячутся алгоритмические ошибки. Правда, на каком-то этапе подобное тестирование может стать нерациональным — вынуждаемая к жизни ошибка оказывается столь редкой и маловероятной, что, возможно, вообще не стоит тратить время на ее исправление.

В некоторых случаях тестирование выполняют с преднамеренно испорченными исходными данными. Так, удары по системам компьютерной безопасности часто наносятся с помощью объемистых или некорректных данных, которые затирают необходимую информацию. Эти слабые места необходимо обнаруживать и исправлять. Некоторые стандартные библиотечные функции легко уязвимы против подобных атак. Например, стандартная функция `gets` не предусматривает никакого способа ограничить размер вводимой строки, поэтому ею вообще не стоит пользоваться. Чтобы не иметь проблем, вместо нее используйте конструкцию `fgets(buf, sizeof(buf), stdin)`. Точно так же не ограничивает длину входной строки вы-

зов функции `scanf("%s", buf)`, поэтому его следует заменить на вызов с явной спецификацией длины строки, например `scanf("%20s", buf)`. В разделе 3.3 было показано, как справиться с этой проблемой в общем случае для буфера произвольной длины.

Любая программа или функция, получающая данные снаружи (прямо или косвенно), должна проверять введенные значения прежде, чем начинать их использовать. Следующая программа взята из учебника. Она считывает целое число, введенное пользователем, и выдает предупреждение, если это число слишком длинное. Цель программы — продемонстрировать решение проблемы с функцией `gets`, хотя предложенное решение не всегда работает.

```
?   #define MAXNUM 10
?
?   int main(void)
?   {
?       char num[MAXNUM];
?
?       memset(num, 0, sizeof(num));
?       printf("Type a number: ");
?       gets(num);
?       if (num[MAXNUM-1] != 0)
?           printf("Number too big.\n");
?       /* ... */
?   }
```

Если введенное число имеет длину десять цифр, оно затрет последний нуль в массиве некоторым ненулевым значением, и теоретически эту ситуацию можно будет распознать после возвращения из функции `gets`. К сожалению, этих мер безопасности недостаточно. Злонамеренный хакер может ввести еще более длинную строку, которая затрет критически важные данные, — например, адрес возврата из функции, — и тем самым заставит программу не возвратиться в тело оператора `if`, а выполнить какую-то другую процедуру, возможно, с разрушительными последствиями. Таким образом, подобная небрежность при проверке исходных данных создает брешь в системе безопасности.

Не подумайте, что это всего лишь искусственный пример из учебника. В июле 1998 года ошибка этого типа была обнаружена в нескольких широко распространенных программах электронной почты. Вот что писали в газете *New York Times*:

> Брешь в системе безопасности возникла из-за так называемого “переполнения буфера”. Программистам обычно вменяется в обязанность включать в свои программы проверку исходных данных на соответствие типов и длин наборов данных. Если набор данных имеет слишком большую длину, он может переполнить “буфер” — участок памяти, выделенный для него. В этом случае программа электронной почты может потерпеть крах, а злонамеренный программист — запустить вместо нее какую-либо вредоносную программу.

Именно так была организована одна из атак знаменитого “Internet-червя” в 1988 г.

Программы, выполняющие синтаксический анализ данных из форм HTML, также могут оказаться уязвимыми для слишком длинных строк, помещаемых в короткие массивы:

```
?    static char *query[1024];
?
?    char *read_form(void)
?    {
?        int qsize;
?
?        qsize = atoi(getenv("CONTENT_LENGTH"));
?        fread(query, qsize, 1, stdin);
?        return query;
?    }
```

В этом коде неявно предполагается, что входная строка никогда не превысит длину 1024 байта. Поэтому данная функция, как и `gets`, подвержена атакам посредством переполнения буфера.

Неприятности могут возникнуть и из-за более простых видов переполнения. Например, незамеченное переполнение целочисленной переменной может привести к катастрофическим последствиям. Рассмотрим следующее распределение памяти:

```
?    char *p;
?    p = (char *) malloc(x * y * z);
```

Если произведение `x * y * z` дает переполнение целочисленного диапазона, то вызов функции `malloc` может, в принципе, привести к выделению буфера памяти вполне разумной, ограниченной длины, однако выражение `p[x]` при этом может ссылаться на участок памяти за его пределами. Предположим, что тип `int` является 16-разрядным, а `x`, `y` и `z` равны 41. Произведение `x*y*z` равно 68921, что при делении на 2^{16} дает остаток 3385. Таким образом, при вызове функции `malloc` будет выделен буфер длиной всего 3385 байт и любое обращение к элементу за пределами этого буфера окажется ошибочным.

Еще одним источником ошибок переполнения является преобразование типов. Простой перехват ошибки может оказаться недостаточным для решения проблемы. Так, в июне 1996 года взорвалась в испытательном полете ракета “Ariane-5”, потому что пакет ее навигационного программного обеспечения был перенесен с “Ariane-4” без должного тестирования. Новая ракета летела значительно быстрее, и в результате некоторые переменные накапливали гораздо бóльшие значения, чем ранее. Вскоре после запуска попытка преобразовать 64-разрядное вещественное число в 16-разрядное целое со знаком привела к переполнению. Ошибка была замечена, но модуль-обработчик принял решение прекратить работу навигационной подсистемы. В результате ракета сошла с курса и взорвалась. По несчастливому совпадению аварийно завершившийся код генерировал инерциальные поправки, необходимые только до взлета; если бы этот модуль отключился в момент запуска, ракета полетела бы нормально.

Если говорить о более повседневных проблемах, часто бывает, что двоичные входные потоки вынуждают аварийное завершение программ, ожидающих на входе текстовых данных, особенно если они работают с 7-разрядным подмножеством

кодировки ASCII. Бывает весьма поучительно (и иногда отрезвляюще) подвергнуть ничего не подозревающую программу, которая ожидает текстовых данных, стрессу в виде двоичного входного потока, например кода скомпилированной программы.

Хорошие тестовые примеры часто можно применить сразу к нескольким программам. Например, всякую программу, считывающую файлы, следует тестировать на пустом файле. Любую программу, читающую и обрабатывающую текстовые строки, необходимо тестировать на очень длинных и пустых строках, а также на входном потоке, вообще не разделенном на строки. Полезно иметь под рукой такой набор файлов, чтобы тестировать любые программы, не создавая всякий раз эти тесты заново. Или же напишите программу, которая по заказу будет сама генерировать необходимый набор тестовых файлов.

Когда Стив Борн (Steve Bourne) разрабатывал командную оболочку Unix (которая со временем стала известна как "среда Борна", "Bourne shell"), он создал каталог из 254 файлов с именами, состоящими из одного символа. Были представлены все символы стандартной кодировки, кроме `'\0'` и косой черты, которые не могут фигурировать в именах файлов Unix. Далее он пользовался этим каталогом для всевозможных тестов, касающихся считывания лексем и поиска по заданным образцам. (Сам каталог с файлами был, разумеется, автоматически сгенерирован программой.) Многие годы после своего создания этот каталог был проклятием для программ, считывающих или отображающих деревья файлов и каталогов, заставляя их терпеть крах или работать некорректно.

Упражнение 6.10. Постарайтесь создать такой файл, который бы привел к аварийному завершению ваш любимый текстовый редактор, компилятор или другую подобную программу.

6.6. Полезные советы

Опытные тестировщики пользуются многими приемами для повышения производительности своего труда. В этом разделе приведены некоторые из наших любимых трюков и подходов.

В программах следует проверять соответствие индексов границам массивов (если язык сам не обеспечивает автоматическую проверку таких вещей); однако до выполнения самого проверяющего кода дело может и не дойти, если размер массива велик по сравнению с типичной длиной исходных данных. Чтобы выполнить проверку, временно сделайте размеры массивов достаточно малыми; так будет легче построить соответствующие тестовые примеры. Аналогичный прием использовался нами в коде для работы с растущими массивами из главы 2, а также в библиотеке CSV из главы 4. Там мы оставили небольшие, тестовые значения размеров, поскольку затраты на инициализацию были незначительными.

Заставьте хэш-функцию возвращать один и тот же хэш-код, чтобы все элементы помещались в одну ячейку таблицы. Это будет эффективным тестом механизма связывания в цепочки, а также проверкой предельного, наихудшего случая.

Если в вашей программе используется собственная функция распределения памяти, напишите такую ее версию, которая через некоторое время перестанет рабо-

тать. Это позволит протестировать код на устойчивость к ошибкам распределения памяти и выхода за пределы выделенных буферов. Следующая версия возвращает NULL после 10 вызовов:

```
/* testmalloc: возвращает NULL после 10 вызовов */
void *testmalloc(size_t n)
{
    static int count = 0;

    if (++count > 10)
        return NULL;
    else
        return malloc(n);
}
```

Прежде чем распространять готовую программу, уберите все тестовые ограничения, влияющие на ее быстродействие и производительность. Как-то раз мы обнаружили одну проблему с быстродействием в профессиональном компиляторе и проследили ее источник до хэш-функции, которая всегда возвращала 0 просто потому, что разработчики оставили на месте тестовый, отладочный код.

Инициализируйте массивы и переменные некоторым характерным значением, а не нулем. В этом случае, если вдруг произойдет выход за пределы массива или обращение к неинициализированной переменной, ошибку будет легче заметить. Например, константу 0xDEADBEEF легко заметить в среде отладчика; процедуры распределения памяти часто используют этот прием для обнаружения неинициализированных данных.

Старайтесь добиться разнообразия в тестовых примерах, особенно если тесты небольшие и запускаются вручную. В противном случае легко попасть в наезженную колею, тестируя все время одно и то же, и при этом не заметить недоработок в другом месте.

Прекратите добавление в программу новых возможностей или даже тестирование существующих, если в ней есть известные, обнаруженные ошибки. Пока эти ошибки не исправлены, они могут существенно влиять на результаты тестирования.

Выводимые при тестировании данные должны содержать значения всех исходных параметров, чтобы тест можно было воспроизвести в точности. Если в программе используются псевдослучайные числа, найдите способ выбрать и вывести значение, инициализирующее их счетчик. Проследите, чтобы исходные параметры и результаты тестирования были представлены в достаточно четкой и ясной форме для их последующего понимания и воспроизведения.

Разумно также обеспечить возможность управления количеством и типами данных, выводимых в ходе работы программы; дополнительные данные могут помочь при тестировании.

Тестируйте программу на нескольких аппаратных платформах, в нескольких выполняющих средах или операционных системах; скомпилируйте ее несколькими разными компиляторами. Такие сочетания позволяют обнаружить особенности, которые не видны другими способами, например, зависимость от порядка байтов в слове и длины целых чисел, восприятие нулевых указателей, символов перевода строки и возврата каретки, специфические свойства библиотек и заголовочных фай-

лов. Тестирование в различных средах также позволяет обнаружить проблемы с объединением компонентов в единый пакет поставки и неявные или непреднамеренные зависимости от среды разработки (подробнее об этом речь пойдет в главе 8).

Тестирование с целью анализа быстродействия рассматривается в главе 7.

6.7. Кто занимается тестированием

Тестирование, выполняемое самим разработчиком программы или тем, у кого есть доступ к исходному коду, иногда называется тестированием методом "белого ящика". Название данного метода вызывает в памяти словосочетание "черный ящик", про который неизвестно, что у него внутри. "Черный ящик" означает, что программист знает, как работать с компонентом, но не знает, как он реализован. Возможно, вместо "белый ящик" лучше было бы сказать "прозрачный ящик".

Очень важно тестировать свой код самостоятельно; не надейтесь, что это сделает за вас какая-нибудь организация или пользователи. Однако легко впасть в заблуждение по поводу того, насколько тщательно вы тестируете, поэтому постарайтесь абстрагироваться от своего, казалось бы, досконального знания кода и думайте не о тривиальных случаях, а об особо сложных. Процитируем рассказ Дональда Кнута (Donald Knuth) о создании тестов для системы форматирования текстов TeX: "Я постарался выработать в себе самый подлый и изощренно коварный настрой, на который только был способен, а затем написал самый гнусный (тестирующий) код, какой только смог придумать; потом я поместил его в еще более извращенные конструкции, которые находились уже на самой грани неприличия". Напомним, что цель тестирования — не продемонстрировать, что программа работает хорошо, а найти в ней ошибки. Поэтому тесты должны быть трудными и коварными, и если они обнаруживают проблему, то следует похвалить свою методику тестирования, а не бить тревогу.

Тестирование методом "черного ящика" означает, что тестирующий ничего не знает о внутреннем устройстве кода и не имеет доступа к исходному тексту. При таком тестировании обнаруживаются самые разные ошибки, поскольку проверке могут подвергнуться самые разные части кода. Начинать тестирование "черного ящика" лучше всего с предельных случаев, а затем переходить к более тяжелым тестам: исходным данным большого объема, искаженным и неожиданным данным и, наконец, некорректным данным, которые программа не должна обрабатывать. Конечно, чтобы убедиться в правильности работы "основной линии" программы, следует задавать и среднестатистические, обычные исходные данные.

На следующем этапе тестирования программа попадает в руки реального пользователя. Новые пользователи находят новые ошибки, поскольку испытывают программу с непредсказуемых сторон. Очень важно протестировать программу таким образом прежде, чем выпускать ее в широкий мир. К сожалению, многие программы выходят в свет без должного тестирования какого бы то ни было вида. Бета-версии программного обеспечения — это как раз попытка отдать программы на растерзание широкому кругу реальных пользователей, чтобы те попробовали их на зуб до выхода финальной версии. Однако выпуск бета-версий не должен подменять собой всестороннее тестирование. Впрочем, по мере того, как программные системы все больше усложняются и разрастаются в размерах, а график их выпуска становится все жест-

че, разработчики все больше поддаются искушению уклониться от тех или иных этапов тестирования.

Довольно трудно тестировать интерактивные программы, особенно если данные вводятся с участием мыши. Некоторые виды тестирования могут выполняться командными сценариями (свойства которых зависят от языка, среды и т.п.). Интерактивные программы желательно писать так, чтобы они могли управляться командными сценариями, имитирующими ввод данных и отклик со стороны пользователя; в этом случае такие программы можно подвергнуть автоматическому тестированию. Один из способов — это записать последовательность операций пользователя, а затем воспроизвести ее автоматически; еще один подход — это встроить в программу язык командных сценариев, позволяющий описывать последовательность и время выполнения команд.

Подумайте также и о том, что тесты сами должны подвергаться тестированию. В главе 5 упоминалась путаница, вызванная некорректной программой тестирования пакета для работы со списками. Если набор регрессионных тестов "заражен" ошибкой, хронические проблемы гарантированы. От результатов тестирования немного проку, если сами тесты содержат ошибки или неточности.

6.8. Тестирование марковской программы

Программа генерирования текстов по марковскому алгоритму (см. главу 3) достаточно сложна, а поэтому требует тщательного тестирования. Она генерирует в целом бессмысленный текст, так что нет возможности сравнивать ее выходные данные с каким-либо эталоном. Как читатель помнит, мы написали несколько версий на разных языках. Наконец, есть еще и та сложность, что программа всякий раз выдает случайный выходной поток, отличающийся от всех предыдущих. Как же нам применить некоторые из уроков этой главы для тестирования такой программы?

Первый набор тестов будет состоять из нескольких крохотных файлов и предназначаться для проверки предельных случаев — корректного генерирования текста на основе исходных данных всего из нескольких слов. Для префиксов длиной 2 мы использовали пять файлов, содержащих соответственно следующие данные (по одному слову в каждой строке):

```
(пустой файл)
a
a b
a b c
a b c d
```

Для каждого из этих файлов на выходе должен получаться текст, идентичный исходному. Эта система тестов сразу же выявила несколько ошибок, связанных с просчетом на единицу при инициализации таблиц, пуске и останове генератора.

Второй тест предназначался для проверки свойств сохранения. Для префиксов из двух слов должно соблюдаться то свойство, что любое отдельное слово, а также любые пары и тройки слов, фигурирующие на выходе, должны содержаться также и в исходном файле. Мы написали программу на языке Awk, которая считывала исходный файл данных в гигантский массив, затем строила массивы всех пар и троек слов,

а потом считывала выходной поток марковской программы в другой массив и сравнивала получившееся:

```
# тест марковской программы: проверка наличия всех слов, пар и троек
#   из выходного потока ARGV[2] во входном потоке ARGV[1]
BEGIN {
    while (getline <ARGV[1] > 0)
        for (i = 1; i <= NF; i++) {
            wd[++nw] = $i   # слова входного потока
            single[$i]++
        }
    for (i = 1; i < nw; i++)
        pair[wd[i],wd[i+1]]++
    for (i = 1; i < nw-1; i++)
        triple[wd[i],wd[i+1],wd[i+2]]++

    while (getline <ARGV[2] > 0) {
        outwd[++ow] = $0    # слова выходного потока
        if (!($0 in single))
            print "unexpected word", $0
    }
    for (i = 1; i < ow; i++)
        if (!((outwd[i],outwd[i+1] in pair))
            print "unexpected pair", outwd[i], outwd[i+1]
    for (i = 1; i < ow-1; i++)
        if (!((outwd[i],outwd[i+1],outwd[i+2]) in triple))
            print "unexpected triple",
                outwd[i], outwd[i+1], outwd[i+2]
}
```

Мы даже не пытались оптимизировать этот тест, удовольствовавшись тем, чтобы сделать программу как можно проще. Чтобы сравнить выходной поток из 10 000 слов со входным потоком из 42 685 слов, программе требовалось около 6-7 секунд, т.е. ненамного больше, чем некоторым версиям тестируемой программы понадобилось для генерирования самого текста. Проверка свойств сохранения позволила найти принципиальную ошибку в Java-версии: программа иногда затирала отдельные позиции в хэш-таблице, поскольку использовала ссылки вместо копирования префиксов.

Данный тест иллюстрирует тот принцип, что бывает легче проанализировать свойства выходных данных, чем сгенерировать их. Например, легче проверить, отсортирован ли файл, чем отсортировать его.

Третий тест по своей природе был статистическим. Исходные данные были такими:

```
a b c a b c ... a b d ...
```

В этой последовательности на десять троек “`a b c`” приходилась одна “`a b d`”. Поэтому в выходном потоке должно было быть примерно в десять раз больше букв `c`, чем `d`, если программа работала с должной стохастичностью. Проверяли мы это, конечно, с помощью программы `freq`.

Статистический тест показал, что ранняя Java-версия программы, ассоциировавшая счетчики с каждым суффиксом, выдавала по 20 букв `c` на каждую `d`, т.е. вдвое больше, чем необходимо. Хорошенько поломав головы, мы наконец сообразили, что

генератор псевдослучайных чисел в Java выдает как положительные, так и отрицательные целые числа. Множитель 2 возник именно потому, что диапазон генерирования был вдвое шире, чем нужно, и нуль, деленный по модулю на счетчик, встречался вдвое чаще. В результате первый элемент списка получал предпочтение. Это как раз и была буква с. Чтобы исправить ошибку, необходимо было брать абсолютное значение перед делением по модулю. Без этого теста ошибка вообще не была бы найдена, потому что на глаз в выходном потоке было все в порядке.

Наконец, мы "скормили" программе обычный английский текст и посмотрели, какую же бессмыслицу она выдаст. Разумеется, это уже проделывалось и на более ранних этапах разработки программы. Но мы не успокоились, добившись того, чтобы программа хорошо обрабатывала корректные входные данные, поскольку на практике бывает всякое. Часто возникает искушение ограничиться только качественными исходными данными, но поддаваться ему не следует; необходимо тестировать программу на любых, самых трудных данных. Чтобы искушение обошло вас стороной, выполняйте тестирование по строгой системе и в автоматизированном режиме.

Все проделанные нами тесты были автоматизированы. Сценарий на командном языке генерировал нужные входные данные, запускал тесты по порядку и регистрировал все аномальные случаи. Этот сценарий был конфигурируемым, так что одни и те же тесты могли применяться ко всем версиям марковской программы. Всякий раз, когда мы вносили в одну из программ набор изменений, все тесты запускались заново для проверки правильности новой версии.

6.9. Резюме

Чем качественнее ваш код с самого начала его существования, тем меньше ошибок в нем будет и тем крепче будет ваша уверенность в том, что проведено полное и всестороннее тестирование. Тестирование предельных случаев по ходу написания кода — это эффективный способ устранения множества мелких и глупых ошибок. Систематическое тестирование — это планомерное нанесение ударов по всем потенциально слабым местам программы. И в этом случае ошибки чаще всего происходят на границах, что можно обнаружить вручную или с помощью программ. Желательно автоматизировать тестирование настолько, насколько это возможно, поскольку автоматические средства не делают случайных ляпов, не устают и не обманывают себя, уверяя, что код работает, если это не так. Регрессионное тестирование позволяет убедиться, что программа по-прежнему выдает правильные ответы, как она делала это раньше, в ее предыдущих версиях. Чтобы локализовать источники проблем, тестируйте код после каждого небольшого изменения, потому что новые ошибки, как правило, находятся в новом коде.

Единственный категорический и абсолютный закон тестирования — это то, что его *нужно делать*.

Дополнительная литература

Один из способов больше узнать о тестировании — это изучить примеры из лучших бесплатно распространяемых программ. В статье Donald Knuth, "The Errors of TEX", *Software — Practice and Experience*, 1985, **19**, **7**, с. 607–685, описываются все ошибки, найденные к тому времени в системе форматирования текстов TEX, а также рассматривается методика тестирования, применяемая автором. Прекрасным примером исчерпывающего набора тестовых примеров может служить тест TRIP для системы TEX. Интерпретатор Perl также поставляется с огромным набором тестовых примеров, который позволяет проверить корректность его работы после компиляции и установки в новой системе. В этот набор входят такие модули, как `MakeMaker` и `TestHarness`, помогающие сконструировать тесты для расширений Perl.

Джон Бентли (Jon Bentley) когда-то написал серию статей для журнала *Communications of ACM*, впоследствии объединенных в сборники *Programming Pearls* и *More Programming Pearls*. Эти сборники были опубликованы издательством Addison-Wesley в 1986 и 1988 гг. соответственно. В этих статьях часто затрагиваются вопросы тестирования, особенно организации сред и оболочек для автоматизации обширных и всесторонних тестов.

7

Быстродействие

Он в обещаньях был могуч, как раньше,
А в исполненьях[1], как теперь, ничтожен.

Вильям Шекспир. “Король Генрих VIII”[2]
(William Shakespear, King Henry VIII*)*

Много лет назад программисты прилагали огромные усилия к тому, чтобы заставить программы работать быстрее, потому что компьютеры были медленными, а машинное время — дорогим. Сейчас цена упала, а скорость процессоров выросла, поэтому нужда в скрупулезной оптимизации быстродействия программ отпала. Так стоит ли вообще беспокоиться о ней?

Да, стоит, но только если проблема действительно существует: программа работает слишком медленно, и есть надежда сделать ее быстрее, при этом сохранив правильность, понятность и надежность. Быстрая программа, дающая неправильный ответ, вряд ли поспособствует экономии времени.

Таким образом, первый принцип оптимизации — это, по возможности, избегать ее. Спросите себя: достаточно ли хорошо работает программа на данный момент? Зная среду и область ее применения, можете ли вы назвать преимущества, которые даст оптимизация ее быстродействия? Программой, написанной в виде домашнего задания в университете, вряд ли кто-нибудь воспользуется хотя бы несколько раз; поэтому ее быстродействие не имеет никакого значения. Так же обстоят дела и с большинством программ для личного пользования, одноразовых утилит, тестовых оболочек, экспериментальных программ и командных сценариев. А вот время выполнения коммерческого продукта или критического компонента, такого как графическая библиотека, может оказаться очень важным, поэтому следует знать, как анализировать и решать проблемы быстродействия.

В каких случаях следует оптимизировать быстродействие программы? Как это нужно делать? Какого результата ожидать? В этой главе рассматриваются следующие вопросы: как заставить программы работать быстрее и как использовать меньше памяти. Обычно быстродействие является более важным, поэтому речь пойдет в основном именно о нем. Вопросы экономии ресурсов (оперативной памяти, дискового пространства) встают реже, но могут оказаться критически важными, поэтому некоторое внимание мы уделим и им.

---

[1] Эпиграф основан на игре слов: *performance* по-английски означает как “исполнение”, так и “быстродействие”. — ***Прим. перев.***

[2] Перевод В.Томашевского.

Как отмечалось в главе 2, наилучшая стратегия программирования состоит в том, чтобы использовать самые простые и ясные алгоритмы и структуры данных из числа тех, что подходят для решения конкретной задачи. Затем следует измерить быстродействие и решить, нужна ли его оптимизация. Включите соответствующие опции компилятора, чтобы получить самый быстрый код; оцените, какие изменения в самой программе повлияют на ее скорость сильнее всего; внесите изменения по одному за раз и снова измерьте быстродействие. При этом сохраните простейшую версию, чтобы было с чем сравнивать оптимизированную программу.

Необходимую часть оптимизации быстродействия составляют измерения, поскольку интуиция и приблизительные прикидки — ненадежные советчики, и ими нельзя подменить точные методы измерения времени и профилирования программ. Оптимизация быстродействия имеет много общего с тестированием, включая такие приемы, как автоматизация, ведение регистрационных записей, регрессионное тестирование для проверки того, что новые изменения сохранили правильность работы программы и не вступили в конфликт с предыдущими усовершенствованиями.

Если тщательно выбирать алгоритмы и писать в хорошем стиле с самого начала, то необходимость в оптимизации может и не возникнуть. В хорошо спроектированной программе проблема быстродействия часто снимается "косметическими" исправлениями, тогда как в плохо написанной программе почти наверняка потребуется существенная реструктуризация.

7.1. Узкие места

Начнем с описания того, как мы убрали узкое место из очень важной программы, работающей в нашей рабочей среде.

Наша входящая электронная почта проходит через систему, именуемую *шлюзом* (*gateway*) и соединяющую нашу внутреннюю сеть с внешней сетью Internet. Сообщения электронной почты из внешнего мира — а это десятки тысяч в день в коллективе из нескольких тысяч человек — сначала поступают в шлюз и только потом передаются во внутреннюю сеть. Такое разделение позволяет изолировать нашу частную сеть от общедоступной сети Internet и представить всех членов коллектива во внешнем мире одним общим именем (именем шлюза).

Одной из функций шлюза является отфильтровка "мусора", или *спама* (*spam*), т.е. непрошеной почты от нежелательных адресатов, в основном содержащей сомнительную рекламу. После успешных предварительных испытаний фильтра мы установили его в качестве постоянной функции для использования всеми пользователями почтового шлюза. И сразу же возникла проблема. Шлюзовая система, уже устаревшая и перегруженная работой, совсем застопорилась, поскольку программа фильтрации работала слишком медленно, отнимая больше времени, чем все остальные операции по обработке сообщений. В итоге почтовые очереди переполнялись, а доставка сообщений задерживалась на целые часы, пока система судорожно пыталась наверстать упущенное.

Это пример самой что ни на есть откровенной проблемы с быстродействием: программа просто-напросто неспособна выполнять свою работу достаточно быстро, и от

этого возникают неудобства у пользователей. Программу следовало заставить работать быстрее.

Говоря несколько упрощенно, спам-фильтр функционирует следующим образом. Каждое входящее сообщение воспринимается как одна строка, и модуль поиска текстовых соответствий проверяет, не содержит ли эта строка фраз из известных спам-сообщений, например, "Make millions in your spare time" ("Заработайте миллионы в свободное время") или "XXX-rated" ("порнография"). Спам-сообщения имеют тенденцию повторяться, так что этот подход вполне практичен; если спам-сообщение не перехватывается один раз, ключевая фраза из него добавляется в список, чтобы в следующий раз оно не прошло.

Ни одна из утилит текстового поиска и сравнения, таких как grep, не обладает должным сочетанием таких качеств, как быстродействие и удобство настройки. Поэтому мы написали специальный спам-фильтр. Первоначальный код был очень прост: в нем каждое сообщение проверялось на наличие любой из фраз заданного списка (образцов).

```
/* isspam: содержит ли mesg любой элемент pat */
int isspam(char *mesg)
{
    int i;
    for (i = 0; i < npat; i++)
        if (strstr(mesg, pat[i]) != NULL) {
            printf("spam: match for `%s'\n", pat[i]);
            return 1;
        }
    return 0;
}
```

Как же сделать эту программу быстрее? Здесь необходимо выполнять поиск в строке, и для этой цели лучше всего подходит библиотечная функция strstr языка C, стандартная и быстродействующая.

С помощью *профилирования* — техники, о которой будет сказано подробнее в следующем разделе, — мы выяснили, что реализация функции strstr неудачна для ее использования в спам-фильтре. Изменяя алгоритм ее работы, ее можно было сделать более эффективной для данной конкретной задачи.

Стандартная реализация strstr выглядела примерно следующим образом:

```
/* простейшая strstr: поиск первого символа с помощью strchr */
char *strstr(const char *s1, const char *s2)
{
    int n;
    n = strlen(s2);
    for (;;) {
        s1 = strchr(s1, s2[0]);
        if (s1 == NULL)
            return NULL;
        if (strncmp(s1, s2, n) == 0)
            return (char *) s1;
        s1++;
    }
}
```

Эту функцию писали, ориентируясь на максимальное быстродействие, и, действительно, в большинстве типичных приложений она работала очень быстро, потому что всю работу в ней выполняли оптимизированные библиотечные средства. Для поиска первого символа строки она вызывает функцию `strchr`, а затем с помощью функции `strncmp` выясняет, соответствует ли продолжение строки заданному образцу. Итак, функция сразу отсеивает бо́льшую часть сообщения, разыскивая только первый символ, а затем быстро проверяет остальное. Почему же этот алгоритм работает недостаточно быстро?

Для этого есть несколько причин и, соответственно, несколько путей совершенствования программы. Во-первых, функция `strncmp` принимает в качестве аргумента длину образца поиска, которая вычисляется функцией `strlen`. Но образцы известны заранее, поэтому их длину не обязательно пересчитывать для каждого сообщения.

Во-вторых, функция `strncmp` имеет сложный внутренний цикл. Она должна не только сравнивать две строки байт за байтом, но еще и проверять завершающий байт (`\0`) в обеих строках, попутно продолжая отсчитывать параметр длины в сторону уменьшения. Поскольку длины всех строк известны заранее (хотя функция `strcmp` об этом не знает), такое усложнение алгоритма не обязательно; мы и так знаем, что счет ведется правильно, так что проверка нулевого байта просто излишня.

В-третьих, функция `strchr` также устроена сложно, поскольку она одновременно разыскивает символ и следит, чтобы не встретился конец строки в виде нулевого байта (`\0`). При каждом конкретном вызове функции `isspam` сообщение имеет фиксированную длину и содержание, поэтому контроль нулевого байта — пустая потеря времени; мы и так знаем, где заканчивается сообщение.

Наконец, хотя функции `strncmp`, `strchr` и `strlen` вполне экономичны сами по себе (в изоляции друг от друга), но расход времени на их вызов сравним с продолжительностью тех операций, которые они выполняют. Было бы эффективнее выполнить всю работу в специальной, тщательно оптимизированной функции типа `strstr`, вообще избежав вызова остальных.

Такого рода проблема — типичный источник неприятностей с быстродействием. Какая-нибудь подпрограмма или интерфейс работает вполне хорошо во всех типичных случаях, однако дает сбои в случаях нетипичных, и к тому же таких, когда данный компонент является центральным для программы. Стандартная функция `strstr` работала отлично для коротких строк и образцов поиска, изменяющихся при каждом вызове, но при длинных фиксированных строках лишний необоснованный расход времени стал уже критическим.

Имея все это в виду, мы переписали функцию `strstr` так, чтобы она выполняла перебор как строки сообщения, так и строки образца поиска одновременно и без вызова каких-либо подпрограмм. Получившаяся реализация обладала предсказуемыми свойствами: она работала несколько медленнее в отдельных случаях, но в целом справлялась с фильтрацией спам-сообщений намного быстрее и, что очень важно, никогда не замедлялась до недопустимо низкой скорости. Для контроля правильности и быстродействия новой версии был разработан набор тестовых данных. Он включал в себя не только простые примеры наподобие поиска слова или предложения, но также и совершенно патологические случаи наподобие поиска одной буквы `x` в строке из тыся-

чи букв e или поиска строки из тысячи x в строке из одной буквы e. Такие задания неизменно вызывали затруднения у стандартной реализации. Подобные экстремальные случаи являются ключевой частью оценки быстродействия.

Итак, в библиотеку была добавлена новая функция strstr, и спам-фильтр заработал примерно на 30% быстрее, что можно считать весьма неплохим результатом исправления всего одной функции.

К сожалению, такой скорости по-прежнему было недостаточно.

При решении задач и проблем важно задавать правильные вопросы. До сих пор мы спрашивали, как быстрее всего найти образец текста в строке. Но фактическая задача состоит в том, чтобы разыскать вхождения строк из большого, заранее известного набора в длинной строке переменной длины и содержания. При такой постановке задачи функция strstr уже не кажется оптимальным решением.

Самый эффективный способ ускорить работу программы — это выбрать для нее более оптимальный алгоритм. Имея четкое представление о задаче, следует подумать, какой же алгоритм лучше всего подходит для ее решения.

Основной цикл выглядит так:

```
for (i = 0; i < npat; i++)
    if (strstr(mesg, pat[i]) != NULL)
        return 1;
```

В нем выполняется npat независимых переборов сообщения; если предположить, что не найдено ни одного соответствия, то каждый байт сообщения перебирается npat раз, что дает strlen(mesg)*npat операций сравнения.

Лучший способ — это обратить циклы и перебирать сообщение один раз во внешнем цикле, параллельно разыскивая в нем все заданные образцы во внутреннем цикле:

```
for (j = 0; mesg[j] != '\0'; j++)
    if (some pattern matches starting at mesg[j])
        return 1;
```

Очередное усовершенствование быстродействия основано на одном простом наблюдении. Чтобы проверить, не совпадает ли текст, начинающийся в позиции j, с каким-либо из образцов, совсем не обязательно перебирать их все — достаточно проверить те, которые начинаются с того же символа, что и mesg[j]. Приблизительно можно считать, что будет выполняться около strlen(mesg)*npat/52 сравнений (поскольку проверяется 26 больших и столько же маленьких букв, в сумме 52). Но частоты употребления букв английского алфавита не одинаковы — слова, начинающиеся с s, встречаются гораздо чаще, чем начинающиеся с x. Поэтому улучшения быстродействия в 52 раза ожидать не приходится, и все же что-то мы выиграем. Фактически мы строим хэш-таблицу, ключом которой является первый символ заданного образца.

Даже после добавления предварительных операций по построению таблицы образцов функция isspam остается достаточно короткой:

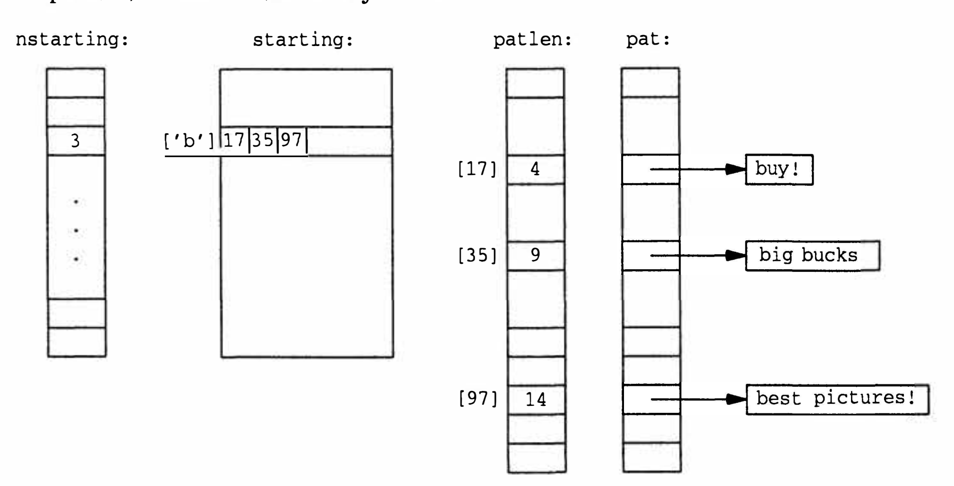
```
int patlen[NPAT];                   /* длина образца */
int starting[UCHAR_MAX+1][NSTART]; /* образцы, нач. с символа */
int nstarting[UCHAR_MAX+1];         /* количество образцов */
...
/* isspam: ищет, встречается ли в mesg любой из образцов */
```

```
int isspam(char *mesg)
{
    int i, j, k;
    unsigned char c;

    for (j = 0; (c = mesg[j]) != '\0'; j++) {
        for (i = 0; i < nstarting[c]; i++) {
            k = starting[c][i];
            if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
                printf("spam: match for `%s'\n", pat[k]);
                return 1;
            }
        }
    }
    return 0;
}
```

Двумерный массив `starting[c][]` содержит для каждого символа `c` индексы тех образцов, которые начинаются с данного символа. В паре с ним используется массив `nstarting[c]`, в котором запоминается, сколько образцов начинается с каждого из символов `c`. Без этих таблиц внутренний цикл выполнялся бы от `0` до `npat`, т.е. порядка тысячи раз; вместо этого он выполняется от `0` до примерно `20`. Наконец, элемент массива `patlen[k]` содержит вычисленный заранее результат операции `strlen(pat[k])`.

На следующем рисунке изображены описанные структуры данных для набора из трех образцов, начинающихся с буквы `b`.



Код для построения этих таблиц выглядит несложно:

```
int i;
unsigned char c;

for (i = 0; i < npat; i++) {
    c = pat[i][0];
    if (nstarting[c] >= NSTART)
        eprintf("too many patterns (>=%d) begin '%c'",
            NSTART, c);
```

```
        starting[c][nstarting[c]++] = i;
        patlen[i] = strlen(pat[i]);
    }
```

В зависимости от состава исходных данных спам-фильтр теперь работает примерно в пять–десять раз быстрее, чем раньше, когда в нем использовалась усовершенствованная функция `strstr`, и от семи до пятнадцати раз быстрее, чем работала его исходная версия. Улучшения в 52 раза достичь не получилось — частично из-за неравномерного распределения букв по частоте употребления, частично из-за большей сложности цикла в новой программе, а частично еще и потому, что по-прежнему приходится выполнять много безрезультатных сравнений строк. Однако спам-фильтр перестал быть узким местом в системе доставки почты. Проблема быстродействия решена.

В остальной части этой главы будут обсуждаться методы обнаружения проблем с быстродействием, вычленения слишком медленного кода и ускорения его работы. Но прежде чем двигаться дальше, стоит еще раз посмотреть на спам-фильтр и подвести итоги, чему же он нас научил. Итак, прежде всего следует убедиться, что проблема действительно заключается в недостаточном быстродействии программы. Все наши усилия пропали бы даром, если бы узким местом системы был не спам-фильтр. Зная, что проблема быстродействия существует, следует воспользоваться техникой профилирования и другими аналогичными методами для изучения поведения программы и локализации конкретного места проявления проблемы. Затем следует убедиться в достаточной общности своего подхода к проблеме; так, мы анализировали всю программу, а не одну только функцию `strstr`, которая первой бросалась в глаза, но отнюдь не являлась основной виновницей. Наконец, мы решили именно ту проблему, которую было нужно, используя более совершенный алгоритм, и убедились, что программа действительно работает быстрее. Как только нужное быстродействие было достигнуто, мы остановились — в конце концов, зачем же перенапрягаться?

Упражнение 7.1. Таблица, в которой одному символу ставится в соответствие набор текстовых образцов, начинающихся с этого символа, дает скачок быстродействия величиною в порядок. Напишите версию функции `isspam`, которая бы использовала в качестве индекса два символа. Какого улучшения быстродействия удается достигнуть в этом случае? Это простые частные случаи структуры данных под названием "TRIE-структура" или "бор", являющейся обобщением дерева поиска. Большинство таких структур данных построено на том или ином компромиссе между объемом и быстродействием.

7.2. Измерение времени и профилирование

Автоматизируйте измерение времени при разработке программ. В большинстве систем имеются команды для измерения времени, затрачиваемого на выполнение программы. В Unix эта команда называется `time`:

```
% time slowprogram
real 7.0
user 6.2
sys 0.1
%
```

После выполнения команды на экране появляются три числа, показывающие время в секундах. Первое число — это "реальное" время, в течение которого работала программа; второе — это "пользовательское" время, затраченное процессором на выполнение программы; третье — это "системное" время, затраченное операционной системой на программу. Если в вашей системе есть аналогичная команда, пользуйтесь ей. В любом случае выдаваемые ею данные будут более информативными, надежными и простыми в получении, чем при измерении времени вручную секундомером. Также не забывайте вести регистрационные записи. При работе над программой, внесении модификаций и выполнении измерений накапливается большой объем данных, которые в один прекрасный день собьют вас с толку. (Какая из двух десятков версий работала на 20% быстрее остальных?) Многие из методов, рассматриваемых в главе о тестировании, можно адаптировать к целям измерения времени и повышения быстродействия. Автоматизируйте выполнение и замеры времени выполнения тестовых примеров и, что еще важнее, не забывайте о регрессионном тестировании, гарантирующем, что вносимые изменения не нарушают правильности работы программы.

Если в системе нет команды замера времени или же необходимо измерить выполнение отдельной функции, то можно написать программу-оболочку (аналогичную оболочке тестирования) для такого замера самостоятельно. В языках C и C++ имеется стандартная функция `clock`, сообщающая, сколько процессорного времени программа затратила к моменту ее вызова. Чтобы измерить длительность выполнения какой-нибудь функции, можно вызвать `clock` перед ней и после нее:

```
#include <time.h>
#include <stdio.h>
    ...
    clock_t before;
    double elapsed;
    before = clock();
    long_running_function();
    elapsed = clock() - before;
    printf("function used %.3f seconds\n",
        elapsed/CLOCKS_PER_SEC);
```

Масштабный множитель `CLOCKS_PER_SEC` характеризует разрешающую способность таймера, сообщаемую функцией `clock`. Если выполнение функции отнимает всего лишь долю секунды, включите ее в цикл, но не забудьте учесть дополнительные затраты времени на организацию цикла, если это имеет значение:

```
before = clock();
for (i = 0; i < 1000; i++)
    short_running_function();
elapsed = (clock()-before)/(double)i;
```

В языке Java имеется класс `Date` с функциями, выдающими обычное время, которое является приближением к времени процессора:

```
Date before = new Date();
long_running_function();
Date after = new Date();
long elapsed = after.getTime() - before.getTime();
```

Функция `getTime` возвращает время в миллисекундах.

Пользуйтесь программами профилирования. Кроме надежного метода измерения времени, наиболее важным инструментом анализа быстродействия является программа или система профилирования. *Профиль* — это подробный отчет о том, на что именно программа тратит свое время выполнения. Некоторые профили состоят из списка функций с указанием количества вызовов каждой и доли общего времени программы, которая затрачена на выполнение этой функции. В других перечисляется, сколько раз выполняется тот или иной оператор. Часто выполняемые операторы отвечают за большую часть общего времени, в то время как никогда не выполняемые могут указывать на бесполезный или должным образом не протестированный код.

Профилирование — это эффективное средство поиска *горячих точек* программы, т.е. функций или фрагментов кода, отнимающих большую часть времени выполнения. Впрочем, и результаты профилирования следует воспринимать критически. Учитывая сложность компиляции и побочных эффектов управления памятью (в том числе кэширования), а также тот факт, что профилирование также влияет на быстродействие программы, статистика в профиле дает лишь приблизительную картину.

В статье, вышедшей в 1971 году, Дональд Кнут (Donald Knuth), автор самого термина "профилирование", писал следующее: "...менее 4 процентов общего объема программы отнимают более половины времени на ее выполнение". Отсюда следует способ применения профилирования. Вначале необходимо обнаружить критические места программы, выполнение которых отнимает больше всего времени, затем оптимизировать их в максимально возможной степени и снова произвести замеры, чтобы проверить, не появится ли на поверхности новая горячая точка. Обычно после одной-двух итераций этого процесса в программе не остается бросающихся в глаза горячих точек.

Профилирование обычно включают специальным ключом или опцией компилятора. Программа выполняется почти как обычно, а затем модуль анализа выдает результаты. В системе Unix этот ключ обычно имеет вид `-p`, а программа профилирования называется `prof`:

```
% cc -p spamtest.c -o spamtest
% spamtest
% prof spamtest
```

В следующей таблице представлен профиль, сгенерированный для специальной версии спам-фильтра. Мы написали ее специально для лучшего понимания работы программы. В ней используются фиксированное почтовое сообщение и фиксированный набор из 217 фраз, разыскиваемый в сообщении 10 000 раз. Эта программа выполнялась на машине MIPS R10000 с тактовой частотой 250 МГц. В ней используется исходная версия алгоритма с функцией `strstr`, вызывающей другие стандартные функции. Результат переформатирован с тем, чтобы помещаться на страницу. Обратите внимание, что объем входных данных (217 фраз) и количество операций поиска (10 000) фигурируют в столбце подсчета количества вызовов "calls", осуществляя тем самым косвенную проверку соответствия алгоритма и программы.

```
12234768552: Total number of instructions executed
13961810001: Total computed cycles
  55.847: Total computed execution time (secs.)
  1.141: Average cycles / instruction
```

secs	%	cum%	cycles	instructions	calls	function
45.260	81.0%	81.0%	11314990000	9440110000	48350000	strchr
6.081	10.9%	91.9%	1520280000	1566460000	46180000	strncmp
2.592	4.6%	96.6%	648080000	854500000	2170000	strstr
1.825	3.3%	99.8%	456225559	344882213	2170435	strlen
0.088	0.2%	100.0%	21950000	28510000	10000	isspam
0.000	0.0%	100.0%	100025	100028	1	main
0.000	0.0%	100.0%	53677	70268	219	_memccpy
0.000	0.0%	100.0%	48888	46403	217	strcpy
0.000	0.0%	100.0%	17989	19894	219	fgets
0.000	0.0%	100.0%	16798	17547	230	__malloc
0.000	0.0%	100.0%	10305	10900	204	realfree
0.000	0.0%	100.0%	6293	7161	217	estrdup
0.000	0.0%	100.0%	6032	8575	231	cleanfree
0.000	0.0%	100.0%	5932	5729	1	readpat
0.000	0.0%	100.0%	5899	6339	219	getline
0.000	0.0%	100.0%	5500	5720	220	_malloc

Бросается в глаза, что функции strchr и strncmp, вызываемые из функции strstr, полностью доминируют в программе по времени выполнения. Итак, мнение Д. Кнута вполне обосновано: выполнение небольшой части программы занимает большую часть общего времени выполнения всей программы. При первом профилировании программы вполне можно обнаружить функцию, которая отнимает более 50% всего времени выполнения, как это и происходит в нашем случае. Здесь легко решить, на чем сосредоточить внимание при оптимизации программы.

Концентрируйтесь на горячих точках. Переписав strstr, мы снова перепрофилировали программу spamtest и обнаружили, что теперь 99,8% всего времени затрачивается на выполнение функции strstr, даже при том, что программа в целом стала работать значительно быстрее. В тех случаях, когда одна-единственная функция является настолько узким местом, есть только два пути: усовершенствовать функцию, применив другой алгоритм, или избавиться от нее вообще, изменив структуру всей программы.

В данном случае мы переписали всю программу. Ниже приведены первые несколько строк профиля программы spamtest с использованием окончательной, быстрой версии функции isspam. Обратите внимание, что общее время выполнения значительно уменьшилось, горячей точкой стала функция memcmp, а isspam теперь потребляет существенную часть ресурсов процессорного времени. Последняя стала значительно сложнее, чем та версия, из которой вызывалась функция strstr, но стоимость этого более чем компенсируется устранением вызовов strlen и strchr, а также заменой strncmp на memcmp, которая выполняет в среднем меньше операций на один байт.

secs	%	cum%	cycles	instructions	calls	function
3.524	56.9%	56.9%	880890000	1027590000	46180000	memcmp
2.662	43.0%	100.0%	665550000	902920000	10000	isspam
0.001	0.0%	100.0%	140304	106043	652	strlen
0.000	0.0%	100.0%	100025	100028	1	main

Поучительно сравнить количество выполнений циклов и вызовов функций в двух приведенных профилях. Заметьте, что `strlen` теперь вызывается не несколько миллионов раз, а всего 652, причем количество вызовов `strncmp` и `memcmp` совпадает. Отметим также, что `isspam`, теперь выполняющая еще и работу функции `strchr`, все же укладывается в намного меньшее количество циклов, чем это делала `strchr`, поскольку на каждом шаге проверяет только образцы, которые реально могут встретиться в тексте. Изучение приведенных чисел может рассказать еще много интересного об особенностях выполнения программы ранее и теперь.

Горячую точку часто можно устранить или хотя бы "охладить" гораздо менее изощренным перепроектированием программы, чем то, к которому мы прибегли в спам-фильтре. Довольно давно профилирование Awk показало, что в ходе регрессионного тестирования одна из функций вызывалась около миллиона раз в следующем цикле:

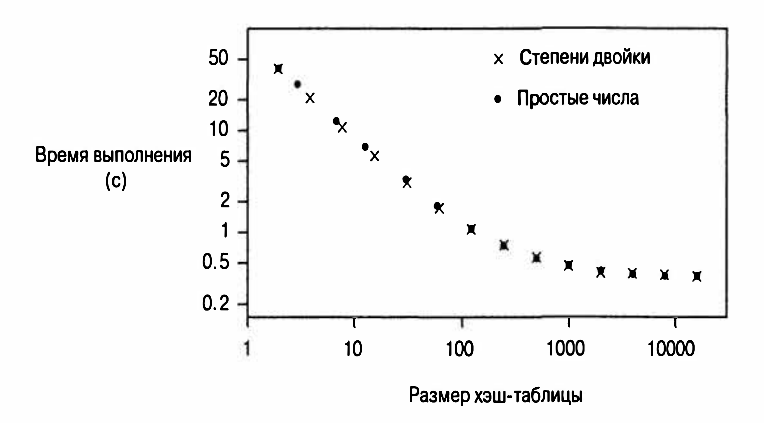
```
?    for (j = i; j < MAXFLD; j++)
?        clear(j);
```

Цикл, в котором очищаются поля перед считыванием следующей строки входных данных, отнимал ни много ни мало 50% всего времени выполнения программы. Константа `MAXFLD`, обозначающая максимально допустимое количество полей в строке данных, была равна 200. Но в большинстве приложений Awk фактическое количество полей не превышало 2 или 3. Таким образом, огромное количество времени тратилось на очистку полей, в которые и так никогда не помещались никакие значения. Замена этой константы ее предыдущим значением дала ускорение примерно на 25%. Чтобы добиться этого повышения быстродействия, понадобилось всего-то заменить верхний предел количества полей:

```
     for (j = i; j < maxfld; j++)
         clear(j);
     maxfld = i;
```

Чертите графики. При измерении быстродействия особо полезны различные графики и схемы, иллюстрирующие результаты таких измерений. По ним можно проследить влияние изменения тех или иных параметров, сравнить алгоритмы и структуры данных, а иногда и выявить случаи неожиданного поведения. Так, графики длин цепочек, построенные для ряда хэш-коэффициентов (см. главу 5), отчетливо продемонстрировали, что некоторые коэффициенты были лучше для использования в хэш-таблице, чем другие.

На следующем графике показано, как размер массива хэш-таблицы влияет на время выполнения C-версии программы `markov` с Книгой Псалмов в качестве исходных данных (42 685 слов, 22 482 префиксов). Мы провели два эксперимента. В одной серии использовались размеры массивов, равные степеням двойки от 2 до 16 384. В другом размеры массивов были равны наибольшим простым числам, меньшим соответствующей ближайшей степени двойки. Нам хотелось посмотреть, влияет ли сколько-нибудь заметным образом на быстродействие программы применение массива с длиной, равной простому числу.



На графике видно, что время выполнения программы с такими входными данными не чувствительно к размеру таблицы, если он превышает 1000 элементов; к тому же никакой заметной разницы между применением степеней двойки и простых чисел не выявлено.

Упражнение 7.2. Вне зависимости от того, есть ли в вашей системе команда `time`, или нет, напишите утилиту измерения времени самостоятельно с применением функции `clock` или `getTime`. Сравните выдаваемые ею результаты с замером времени по секундомеру. Как влияют на измерение времени системные операции?

Упражнение 7.3. В первом из приведенных выше профилей функция `strchr` вызывалась 48 350 000 раз, тогда как `strncmp` — только 46 180 000. Объясните эту разницу.

7.3. Стратегия ускорения

Прежде чем оптимизировать быстродействие программы, убедитесь, что она действительно работает недостаточно быстро. Чтобы выяснить, на что же тратится время, воспользуйтесь утилитами измерения времени и профилирования. Зная, что в действительности происходит, прибегните к одной из известных стратегий оптимизации. Некоторые из них описаны ниже в порядке убывания важности.

Выберите лучший алгоритм или структуру данных. Наиболее важным фактором, влияющим на быстродействие программы, является выбор алгоритмов и структур данных. Между эффективными и неэффективными алгоритмами бывает огромная разница. В нашем спам-фильтре замена структуры данных привела к ускорению в десять раз; возможна и еще бóльшая оптимизация, если новый алгоритм уменьшает порядковую оценку количества операций, например, с $O(n^2)$ на $O(n \log n)$. Эта тема рассматривалась в главе 2, так что здесь на ней останавливаться не будем.

Удостоверьтесь, что сложность алгоритма действительно имеет под собой основания; в противном случае за этой сложностью может скрываться дефект быстродействия. Например, следующий алгоритм перебора строки на первый взгляд кажется линейным:

```
?    for (i = 0; i < strlen(s); i++)
?        if (s[i] == c)
?            ...
```

Но на самом деле он имеет квадратичный порядок: если в строке n символов, то при каждом вызове функции `strlen` перебираются все эти n символов и при этом весь цикл также выполняется n раз.

Включите оптимизирующую компиляцию. Одна из модификаций, которая не требует никаких усилий, но может дать существенное улучшение быстродействия, — это включить все опции оптимизации, которые предлагает компилятор. Современные компиляторы позволяют программисту практически обойтись без выполнения большинства локальных оптимизаций вручную, поскольку достаточно хорошо делают это автоматически.

По умолчанию компиляторы C и C++ не выполняют почти никакой оптимизации. Одна из опций компилятора активизирует оптимизатор (хотя более точным термином был бы "реорганизатор"). Возможно, стоило бы сделать его включаемым по умолчанию, кроме тех случаев, когда оптимизация вносит путаницу в отладку исходного кода. Поэтому программистам следует включать оптимизацию тогда, когда они полагают, что отладка закончена.

Оптимизирующая компиляция повышает быстродействие по-разному: от нескольких процентов до двух раз. Однако бывает и так, что программа от подобной оптимизации только замедляется, поэтому следует тщательно измерить степень улучшения до предоставления программы пользователю. Мы сравнили оптимизированный и неоптимизированный код на примере одной-двух версий спам-фильтра. Для тестового примера с применением окончательной версии алгоритма поиска и сравнения исходное время выполнения составляло 8,1 секунды, а при включенной оптимизации — 5,9 секунды, что составило улучшение почти на 25%. С другой стороны, версия со стандартной функцией `strstr` вообще не поддалась оптимизации, потому что эта функция уже была оптимизирована при ее помещении в библиотеку. Оптимизация применяется только к коду, компилируемому в данный момент, но не к тому, который берется из системных библиотек. Однако в некоторых компиляторах все-таки есть глобальные оптимизаторы, анализирующие всю программу на возможность потенциального улучшения быстродействия. Если такой компилятор имеется в вашей системе, испытайте его; он может помочь вам сократить работу по оптимизации кода.

Следует иметь в виду одно обстоятельство: чем интенсивнее компилятор выполняет оптимизацию, тем более вероятно, что он может внести ошибку в скомпилированную программу. Поэтому после включения оптимизации следует еще раз выполнить регрессионное тестирование, как и в случае любых других модификаций.

Выполняйте тонкую настройку кода. Если объем обрабатываемых данных очень велик, большое значение имеет правильный выбор алгоритма. Более того, алгоритмические усовершенствования сохраняются при переходе в другие системы, к другим компиляторам и языкам. Но если уже выбран наилучший возможный алгоритм, а проблема быстродействия остается, то следующее, что следует попробовать, — это выполнить *тонкую настройку* кода, т.е. попытаться изменить организацию циклов и выражений так, чтобы они выполнялись как можно быстрее.

Та версия isspam, которую мы продемонстрировали в конце раздела 7.1, не подвергалась такой настройке. Здесь же будет показано, как можно достичь дальнейшего улучшения, реорганизовав цикл. Напомним, как он выглядел первоначально:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    for (i = 0; i < nstarting[c]; i++) {
        k = starting[c][i];
        if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
            printf("spam: match for '%s'\n", pat[k]);
            return 1;
        }
    }
}
```

Эта начальная версия, скомпилированная с включенной оптимизацией, выполняется за 6,6 секунды на нашем тестовом примере. В условии внутреннего цикла фигурирует индекс массива (nstarting[c]), значение которого фиксировано на протяжении одного прохода внешнего цикла. Поэтому его пересчета можно избежать, сохранив это значение в локальной переменной:

```
for (j = 0; (c = mesg[j]) != '\0'; j++) {
    n = nstarting[c];
    for (i = 0; i < n; i++) {
        k = starting[c][i];
        ...
```

В результате время выполнения падает до 5,9 секунды, что примерно на 10% быстрее. Именно такая степень оптимизации обычно достигается тонкой настройкой. Имеется еще одна переменная, использование которой можно реорганизовать, — элемент starting[c] также фиксирован. На первый взгляд, вынесение его вычисления за пределы цикла может еще ускорить процесс, но в нашем тесте никакой заметной разницы обнаружено не было. Это также характерно для тонкой настройки: что-то поддается оптимизации, что-то — нет, и следует проводить точные измерения, чтобы обнаружить эту разницу. Кроме того, результаты такой настройки могут отличаться на различных платформах и с различными компиляторами.

В спам-фильтр можно внести и еще одно усовершенствование. Во внутреннем цикле проверяется наличие в строке целого заданного образца текста, но сам алгоритм уже гарантирует совпадение первых символов. Поэтому мы попытались выполнить тонкую настройку, начиная сравнение в функции memcmp на один байт дальше. Настройка дала ускорение около 3%, что хотя и немного, но зато потребовало изменения всего трех строк кода, одна из которых выполняет предварительные вычисления.

Не оптимизируйте то, что не имеет значения. Иногда тонкая настройка не дает никакого результата, потому что применяется там, где оптимизировать нечего. Убедитесь, что ваша оптимизация действительно применяется к коду, выполнение которого отнимает много времени. Следующая история не вполне достоверна, но мы ее все же расскажем. Одну из систем ныне несуществующей компании анализировали с помощью аппаратного измерителя быстродействия и обнаружили, что около 50% всего времени она выполняет одну и ту же последовательность нескольких инструкций. Инженеры встроили в систему специальную инструкцию, объеди-

няющую в себе функции той последовательности, перезапустили новую систему и обнаружили, что никакой разницы с предыдущей нет. В итоге оказалось, что они оптимизировали холостой цикл операционной системы!

Как много времени и усилий позволительно тратить на оптимизацию быстродействия программы? Основной критерий состоит в том, чтобы вносимые изменения окупились в будущем. Ориентировочно можно сказать так: рабочее время, потраченное на настройку быстродействия, не должно быть больше, чем время, впоследствии сэкономленное этой настройкой за срок существования программы. В соответствии с этим правилом алгоритмическое усовершенствование программы `isspam` стоило свеч: оно заняло один рабочий день, а сэкономило (и продолжает экономить) несколько часов каждый день. Устранение индексации массива во внутреннем цикле менее заметно, однако и его стоило проделать, поскольку программа в течение длительного времени обслуживает большой коллектив пользователей. Оптимизация общественно полезных программ наподобие спам-фильтра или библиотек окупается почти всегда, тогда как оптимизация тестовых одноразовых программ — почти никогда. Если программа должна проработать как минимум год, оптимизируйте в ней все, что только можете. Оптимизацию стоит выполнить, даже если программа уже работает месяц, но вы обнаружили способ ускорить ее работу еще на 10%.

Коммерческие и другие общедоступные программы, испытывающие конкуренцию со стороны своих аналогов, такие как игры компиляторы, электронные таблицы, системы управления базами данных, тоже попадают в эту категорию, поскольку успехом у пользователя чаще всего пользуются наиболее быстрые и эффективные из них. По крайней мере, так отмечается в опубликованных сравнительных характеристиках.

Важно измерять время выполнения программ по мере внесения изменений, чтобы убедиться, что программа постепенно совершенствуется. Иногда два изменения, которые сами по себе улучшают работу программы, накладываясь, нейтрализуют эффект друг друга. В этом случае измерение времени может давать настолько неожиданные результаты, что становится трудно делать какие-то заключения о влиянии вносимых изменений на быстродействие.

Даже в однопользовательских системах время выполнения подвержено непредсказуемым флуктуациям. Если разброс измерений внутреннего таймера (или, по крайней мере, тех значений, которые сообщает система) составляет 10%, то изменения, дающие оптимизацию в пределах этих же 10%, трудно отличить от фонового шума.

7.4. Настройка кода

Существует много способов снизить затраты времени, если в программе обнаружена горячая точка. В этом разделе приведены некоторые рекомендации, которыми следует пользоваться аккуратно, неизменно выполняя регрессионное тестирование после каждой модификации, чтобы убедиться, что код по-прежнему работает правильно. Имейте в виду, что хорошие компиляторы выполняют некоторые из этих оптимизаций самостоятельно, и нередко им можно только помешать, усложняя программу собственными дополнениями. Короче говоря, что бы вы ни делали с программой, выполняйте измерения, чтобы убедиться в эффективности вносимых изменений.

Объединяйте вычисление одинаковых подвыражений. Если в фрагменте кода несколько раз вычисляется одно и то же сложное выражение, проделайте вычисления один раз и запомните результат. Например, в главе 1 демонстрировался макрос, который вычислял расстояние, причем в нем дважды вызывалась функция `sqrt`. По сути, вычислялось следующее выражение:

```
?     sqrt(dx*dx + dy*dy) + ((sqrt(dx*dx + dy*dy) > 0) ? ...)
```

Следовало бы вычислить квадратный корень один раз и воспользоваться результатом в двух местах.

Если вычислительная операция выполняется в цикле, но не зависит от переменных параметров цикла, вынесите эту операцию за его пределы. Мы проделали это при реорганизации следующего оператора:

```
for (i = 0; i < nstarting[c]; i++) {
```

После реорганизации этот фрагмент стал выглядеть так:

```
n = nstarting[c];
for (i = 0; i < n; i++) {
```

Заменяйте дорогостоящие операции на более простые. Одним из видов оптимизации является упрощение операций. В минувшие времена это, например, означало замену умножения сложением или сдвигом, хотя в наши дни это редко окупается. Однако деление и взятие остатка выполняются намного медленнее, чем умножение, поэтому определенного улучшения можно добиться, заменив деление умножением на обратную величину, а взятие остатка — операцией наложения битовой маски, если делитель является степенью двойки. Замена обращения к массиву по индексу с использованием указателя в C или C++ может дать некоторое ускорение, хотя большинство компиляторов и так делают это автоматически. Замена вызова функции более простой вычислительной операцией все еще может иметь смысл. Расстояние между точками на плоскости определяется формулой `sqrt(dx*dx+dy*dy)`, поэтому решение вопроса о том, какая из двух пар точек расположена на большем расстоянии, обычно включает в себя вычисление двух квадратных корней. Однако то же самое решение можно принять, сравнивая квадраты расстояний. Следующий оператор дает тот же результат, что и сравнение квадратных корней:

```
if (dx1*dx1+dy1*dy1 < dx2*dx2+dy2*dy2)
    ...
```

Еще один пример подобной оптимизации встречается в системах поиска заданных текстовых образцов наподобие нашего спам-фильтра или утилиты `grep`. Если такой образец начинается с литерального символа, то вначале выполняется быстрый поиск во входном тексте по этому первому символу. Если символа в тексте нет, то более сложный механизм поиска всего образца вообще не активизируется.

Разворачивайте или устраняйте циклы. При организации и выполнении циклов затрачивается некоторое дополнительное время и ресурсы системы. Если тело цикла достаточно короткое и выполняется лишь несколько раз, то бывает быстрее просто выписать все итерации одна за другой в исходном коде. Рассмотрим следующий пример:

```
for (i = 0; i < 3; i++)
    a[i] = b[i] + c[i];
```

Его можно переписать в таком виде:

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
```

Такая запись устраняет дополнительные расходы по организации цикла, в частности, по его ветвлению, которое может замедлить современный процессор, прерывая непрерывный поток вычислений.

Если цикл имеет большую длину, аналогичное преобразование можно использовать для сокращения количества итераций:

```
for (i = 0; i < 3*n; i++)
    a[i] = b[i] + c[i];
```

После реорганизации цикл будет выглядеть так:

```
for (i = 0; i < 3*n; i += 3) {
    a[i+0] = b[i+0] + c[i+0];
    a[i+1] = b[i+1] + c[i+1];
    a[i+2] = b[i+2] + c[i+2];
}
```

Заметьте, что данный конкретный прием срабатывает только тогда, когда продолжительность цикла кратна количеству операторов в его теле. В противном случае необходим дополнительный код для состыковки в конце, а это провоцирует ошибки и снижение быстродействия.

Кэшируйте часто используемые данные. Кэшированные значения нет необходимости пересчитывать заново. При кэшировании используется преимущество *локальности*, т.е. тенденции программ (как и людей) снова и снова обращаться к недавно использованным или близлежащим объектам и данным в противовес более старым или отдаленным. В вычислительной технике кэш-память применяется очень широко; установка дополнительной кэш-памяти на компьютер может существенно ускорить работу системы. То же справедливо и для программ. Например, Web-браузеры кэшируют страницы и графические изображения, чтобы сэкономить на медленной пересылке данных по Internet. В программе предварительного просмотра печатного документа, которую мы написали много лет назад, выполнялся поиск в таблице таких неалфавитных специальных символов, как $^1/_2$. Как показали замеры быстродействия, большей частью применение специальных символов состояло в вычерчивании линий из длинных цепочек одного и того же символа. Кэширование всего одного — самого последнего — из использованных символов сразу же сделало программу существенно быстрее на большинстве типичных наборов входных данных.

Лучше всего сделать так, чтобы операция кэширования была невидима внешнему пользователю и внешним модулям. В этом случае остальная часть программы никак не будет затронута этой операцией, за исключением общего повышения быстродействия. Так, в случае программы предварительного просмотра печати интерфейс функции вывода символов никак не менялся; он всегда выглядел следующим образом:

```
drawchar(c);
```

В исходной версии drawchar выполнялся вызов show(lookup(c)). В реализации с кэшированием использовались дополнительные внутренние статические переменные, в которых запоминался предыдущий символ и его код:

```
if (c != lastc) { /* update cache */
    lastc = c;
    lastcode = lookup(c);
}
show(lastcode);
```

Пишите свои функции распределения памяти. Часто единственной горячей точкой программы является распределение памяти, проявляющееся в виде большого количества вызовов функции malloc или обращений к операторам new. Если в большинстве запросов требуются блоки одного и того же размера, то можно добиться существенного ускорения путем замены общих функций распределения памяти специальными. Такая специальная функция выполняет один вызов malloc, получает один большой массив блоков, а затем раздает их по одному по мере необходимости, что в итоге обходится проще и дешевле. Освобожденные блоки возвращаются в список, чтобы их можно было быстро задействовать снова.

Если размеры запрашиваемых блоков близки, но не одинаковы, можно сэкономить время за счет некоторого увеличения объема памяти. Для этого следует выделять фиксированный максимальный объем памяти для каждого блока. Этот подход бывает эффективен при работе с короткими строками. В этом случае они помещаются в буферы фиксированного размера, равного длине наибольшей строки.

В некоторых алгоритмах, в которых сначала запрашивается сразу много буферов памяти, а потом они все сразу освобождаются, можно использовать стековую модель распределения памяти. В этом случае модуль распределения памяти получает в свое распоряжение один большой массив и рассматривает его как стек, одним махом помещая в него новые объекты в начале и извлекая освобождаемые в конце программы. В некоторых библиотеках C для этой цели имеется функция alloca, хотя она не относится к числу стандартных. В ней в качестве источника памяти используется локальный стек модуля, и все размещенные объекты освобождаются, как только функция, вызвавшая alloca, возвращает управление.

Буферизуйте ввод и вывод. Буферизация объединяет операции в пакет, так что часто выполняемые манипуляции с данными организуются по максимально эффективному графику, а очень затратные операции выполняются только в случае крайней необходимости. Таким образом, затраты на выполнение ввода-вывода равномерно распределяются по целым наборам данных. Например, когда в программе на C вызывается функция printf, символы помещаются в буфер, но не передаются в операционную систему до тех пор, пока буфер не наполняется или не сбрасывается в поток вывода принудительно. В свою очередь, операционная система также может задержать запись данных на диск. Недостаток этого подхода состоит в том, что необходимо очистить буфер вывода, чтобы сделать данные видимыми пользователю. В самом худшем случае информация, все еще находящаяся в буфере, будет потеряна, если программа завершится аварийно.

Обрабатывайте особые случаи отдельно. Путем обработки объектов одного размера в отдельном модуле кода функции специального распределения памяти помогают снизить затраты времени и ресурсов в функциях общего назначения, а также иногда уменьшают фрагментацию памяти. В графической библиотеке для системы Inferno базовая функция прорисовки была сначала написана как можно проще и очевиднее. Как только эта версия заработала, по одной стали добавляться оптимизации для различных специальных случаев (выделенных путем профилирования). Простейшая версия сохранилась, так что оптимизированные варианты всегда можно было сравнить с ней. В конце концов авторы оставили оптимизацию всего лишь нескольких случаев, потому что динамическое распределение вызовов функции прорисовки сильно склонялось в сторону отображения символов, а писать хитроумный код для всех случаев оказалось нерационально.

Выполняйте предварительные вычисления. Иногда программу можно сделать быстрее, вычисляя некоторые величины заранее, чтобы они были уже под рукой в момент, когда они понадобятся. Этот принцип применялся в спам-фильтре, в котором величина `strlen(pat[i])` вычислялась заранее и помещалась в позицию массива `patlen[i]`. Если графической системе необходимо постоянно пересчитывать значения математической функции наподобие синуса, но лишь для дискретного набора аргументов, например целого числа градусов, то быстрее будет заранее просчитать таблицу из 360 позиций (или предоставить ее в виде исходных данных) и обращаться к ней по индексам. Это типичный пример экономии времени за счет используемого объема памяти. Существует множество возможностей для замены кода данными или выполнения вычислений при компиляции для экономии времени выполнения, а иногда и ресурсов памяти. Например, функции семейства `ctype` наподобие `isdigit` почти всегда реализуются в виде обращения к таблицам битовых флагов, а не с помощью сравнений и других вычислительных операций.

Используйте приближенные значения. Если точность вычислений не имеет решающего значения, используйте типы данных меньшей точности, т.е. разрядности. В старых или небольших системах, как и в тех, где вещественнозначные операции выполняются программно, а не аппаратно, вещественная арифметика одинарной точности часто работает быстрее, чем арифметика двойной точности. Поэтому для экономии времени можно воспользоваться типом `float` вместо `double`. В некоторых современных программах для работы с графикой используется аналогичный прием. Стандарт IEEE по вещественной арифметике предписывает "корректную обработку потери значимости" при приближении результата вычислений к нижней границе представляемых значений. Однако это обходится недешево. При обработке изображений подобная операция не имеет никакого смысла. Целочисленные функции вычисления синуса и косинуса — это еще один пример приближенных вычислений этого рода.

Переписывайте части программы на языке более низкого уровня. Языки более низкого уровня имеют тенденцию к большему быстродействию, хотя это достигается за счет рабочего времени программиста. Поэтому переписывание отдельных критических модулей программы, написанной на C++ или Java, на языке C или

замена интерпретируемого сценария скомпилированной программой может значительно улучшить быстродействие.

Иногда можно получить существенное повышение быстродействия, используя машинно- или системно-зависимый код. Это самое крайнее средство, и прибегать к нему без нужды не следует, потому что из-за этого ухудшается переносимость и затрудняется доработка и поддержка программы. Почти всегда те модули, которые желательно переписать на языке ассемблера, являются сравнительно небольшими функциями, легко встраиваемыми в какую-нибудь библиотеку. Типичными примерами являются `memset`, `memmove` и графические операции. Основной подход состоит в следующем. Необходимо написать код так чисто и правильно, как только возможно, целиком на языке высокого уровня и полностью протестировать его, как это описано в главе 6 для функции `memset`. Это будет переносимая версия, работающая на любых платформах, хотя и медленно. Таким образом, при переходе в новую среду можно будет начинать с гарантированно рабочей версии программы. Написав версию с применением языка ассемблера, протестируйте ее на полную эквивалентность переносимой версии. Если случаются ошибки, под подозрение всегда попадает непереносимая часть кода, и здесь полезно иметь эталонную версию для сравнения.

Упражнение 7.4. Один из способов повысить быстродействие такой функции, как `memset`, — это заставить ее записывать данные словами, а не байтами. Этот способ лучше совместим с аппаратурой и может снизить дополнительные затраты при организации циклов в четыре–восемь раз. Недостаток этого подхода состоит в том, что если блок назначения не выровнен по границе слова или его длина не кратна длине слова, то возникают различные побочные эффекты. Напишите версию этой функции, оптимизированную данным способом. Сравните ее быстродействие с существующей библиотечной версией и простым циклом побайтовой записи.

Упражнение 7.5. Напишите функцию распределения памяти `smalloc` для строк C, в которой бы использовался специальный модуль распределения для небольших строк, но прямой вызов `malloc` — для больших. Необходимо будет определить структуру для представления строк в каждом случае. Как принять решение, в каком случае переключаться с `smalloc` на `malloc`?

7.5. Оптимизация использования памяти

В свое время память была самым драгоценным вычислительным ресурсом. Ее всегда не хватало, и при попытках выжать максимальный результат из того малого, что имелось в наличии, было написано немало плохих программ. В качестве характерного примера часто приводят пресловутую "проблему 2000 года": когда памяти действительно ни на что не хватало, даже экономия двух байт на представлении цифр 19 казалась существенной. Но независимо от того, действительно ли причиной проблемы была попытка сэкономить память (а это могло быть вызвано и простой привычкой опускать столетия в датах в повседневной жизни), это хороший пример близорукой, недальновидной оптимизации. В любом случае, времена изменились, и теперь как оперативная, так и внешняя память стоят удивительно дешево. Поэтому

первый принцип оптимизации использования памяти должен быть таким же, как и в отношении быстродействия, — постарайтесь вообще ее не делать.

Однако бывают и ситуации, когда эффективность использования памяти все-таки имеет значение. Если программа не помещается в оперативную память, то отдельные ее части будут храниться в виртуальной памяти на диске, что сделает быстродействие недопустимо низким. Это можно часто видеть при установке новых версий программ; после обновления они тут же начинают расходовать память. Суровая реальность такова, что вслед за модернизацией программного обеспечения часто приходится покупать дополнительную память.

Экономьте память, используя наименьший возможный тип данных. Первый шаг к эффективности использования памяти — это оптимизировать работу с уже задействованными ресурсами, например, применяя самые короткие типы данных, пригодные для решения задачи. Например, тип `int` заменяется типом `short`, если в него помещаются рабочие данные, такие как координаты в системах работы с двумерной растровой графикой, поскольку 16 бит наверняка хватит для работы с любым возможным диапазоном экранных координат. Тип `double` также иногда заменяется типом `float`; потенциальная проблема состоит в потере точности, поскольку для чисел типа `float` обычно запоминается всего 6 или 7 значащих десятичных цифр. В этих и аналогичных случаях могут понадобиться также и другие изменения, самые заметные из которых — замена спецификаций формата в вызовах функций `printf` и особенно `scanf`.

Этот подход применим и к представлению логической информации. Ее можно кодировать одним байтом или отдельными битами, по возможности одним. Не следует пользоваться битовыми полями С и C++; они очень плохо переносимы и приводят к генерированию слишком громоздкого кода. Вместо этого следует инкапсулировать требуемые операции в функциях, которые бы устанавливали или запрашивали отдельные биты машинного слова или массива слов с помощью операций поразрядного маскирования и сдвига. Следующая функция возвращает группу последовательных битов из середины слова:

```
/* getbits: получает n бит из позиции p */
/* биты нумеруются с 0, начиная с младших */
unsigned int getbits(unsigned int x, int p, int n)
{
        return (x >> (p+1-n)) & ~(~0 << n);
}
```

Если такие функции работают слишком медленно, их можно усовершенствовать с помощью методов, описанных ранее в этой главе. В языке C++ для представления доступа к битам в виде обращения по индексу можно использовать перегрузку операций.

Не храните данные, которые легко вычислить заново. Подобные изменения имеют локальный и частный характер; они аналогичны тонкой настройке кода. Более глобальные усовершенствования обычно связаны с перестройкой структур данных, иногда вместе с изменением алгоритма. Вот один пример. Много лет назад к одному из нас обратился коллега, который пытался выполнить вычисления над такой большой матрицей, что только для того, чтобы она поместилась в память, приходилось выключать компьютер и загружать урезанную операционную систему.

Ему хотелось знать, нет ли какой-нибудь разумной альтернативы, потому что работа в подобном режиме походила на кошмар. Мы спросили, что представляет собой эта матрица, и оказалось, что она содержит целочисленные значения, причем большей частью нули. Фактически ненулевыми были менее пяти процентов элементов матрицы. Сразу же возникла идея представления матрицы, в котором хранились бы только ненулевые элементы, а каждое обращение по индексам `m[i][j]` заменялось бы вызовом функции `m(i,j)`. Существует несколько способов хранить подобные данные. Самый простой из них — это массив указателей, по одному на каждую строку, которые указывают на компактный массив из номеров столбцов и соответствующих чисел. В этом представлении на каждый ненулевой элемент в среднем затрачивается больше памяти, но в целом матрица занимает значительно меньше места. Хотя обращение к отдельным элементам несколько замедляется, в целом процедура становится намного быстрее, чем перезагрузка операционной системы. Короче говоря, наш коллега применил предложенный способ и остался вполне доволен.

Мы применили подобный способ для решения современного варианта той же проблемы. В системе проектирования радиосетей необходимо было представить ландшафтную информацию и интенсивность радиосигнала в большом географическом регионе (от 100 до 200 км по стороне) с разрешением около 100 метров. Хранить подобную информацию в большом прямоугольном массиве означало превысить объем памяти, имеющийся в системе, и вызвать недопустимые затраты времени на ее виртуализацию. Но оказалось, что в довольно больших регионах ландшафт и интенсивность сигнала имеют тенденцию сохраняться постоянными, так что проблема оказалась решаемой с помощью иерархического представления, в котором регионы с одними и теми же показателями объединялись в одну ячейку.

Вариации на эту тему весьма распространены, как и конкретные представления данных, но все их объединяет общая идея: хранить тривиальные, общие значения неявно или в компактной форме, пожертвовав несколько бо́льшим временем и ресурсами на обработку остальных значений. Если бо́льшую часть данных действительно составляют общие значения, эта стратегия имеет успех.

Программу следует организовать так, чтобы конкретное представление данных сложного типа было инкапсулировано в классе или наборе функций, оперирующих с закрытыми типами или наборами данных. Эта мера предосторожности необходима для того, чтобы остальная часть программы не была затронута при возможном изменении представления данных.

Иногда проблемы экономии памяти появляются также и при внешнем представлении информации — как преобразовании, так и хранении. В целом всегда лучше хранить информацию в текстовом виде, когда возможно, чем в двоичном представлении какого бы то ни было вида. Текст лучше переносится между системами, легко читается, поддается обработке любыми средствами. У двоичных представлений нет ни одного из подобных преимуществ. Аргументы в пользу двоичных представлений обычно основываются на преимуществах "скорости обработки", но к ним следует относиться скептически, поскольку разница в скорости между текстовой и двоичной формой может оказаться совсем не такой большой.

Некоторая экономия часто достигается за счет жертвования временем выполнения. Так, в одном приложении необходимо было передавать большое изображение из одной программы в другую. Изображения в простом формате (PPM) обычно

занимали около мегабайта, поэтому нам показалось, что будет быстрее закодировать их для пересылки в сжатом формате GIF, в котором файлы занимали около 50 килобайт. Но кодирование и декодирование GIF-формата отнимало столько же времени, сколько экономилось путем передачи более короткого файла, так что никакой экономии, по сути, не достигалось. Код для обработки GIF-формата имел длину около 500 строк, а для работы с PPM-форматом — примерно 10 строк. Поэтому для удобства работы с программой идея с GIF-кодированием была отброшена, и в приложении продолжал использоваться исключительно формат PPM. Разумеется, мы пришли бы совсем к другим выводам, если бы файл нужно было пересылать по медленной сети. Тогда формат GIF имел бы большое преимущество.

7.6. Некоторые оценки

Заранее оценить, насколько быстро будет работать программа, очень трудно, а еще труднее оценить затраты на конкретные операторы языка или машинные инструкции. Тем не менее довольно просто построить *модель стоимости* языка или системы, которая даст, по крайней мере, грубое представление о продолжительности выполнения важнейших операций.

В традиционных языках программирования часто используется следующий подход: пишется программа, измеряющая время выполнения типичных, представительных фрагментов кода. Здесь имеются некоторые операционные трудности, например, получение надежно воспроизводимых результатов и отфильтровывание накладных затрат времени. Но тем не менее можно получить полезные оценки без особых усилий. Например, у нас есть программа для составления модели стоимости языков C и C++, которая оценивает затраты на отдельные операторы. Для этого операторы заключаются в цикл и выполняются много миллионов раз, а затем вычисляется среднее время. На компьютере MIPS R10000 с тактовой частотой 250 МГц были получены следующие данные, в которых длительность операций указана в наносекундах:

```
Int Operations
  i1++                                  8
  i1 = i2 + i3                         12
  i1 = i2 - i3                         12
  i1 = i2 * i3                         12
  i1 = i2 / i3                        114
  i1 = i2 % i3                        114

Float Operations
  f1 = f2                               8
  f1 = f2 + f3                         12
  f1 = f2 - f3                         12
  f1 = f2 * f3                         11
  f1 = f2 / f3                         28

Double Operations
  d1 = d2                               8
  d1 = d2 + d3                         12
  d1 = d2 - d3                         12
  d1 = d2 * d3                         11
  d1 = d2 / d3                         58
```

```
Numeric Conversions
  i1 = f1                                    8
  f1 = i1                                    8
```

Целочисленные операции — достаточно быстрые, за исключением деления и взятия остатка. Операции над вещественными числами с плавающей точкой, как оказалось, почти настолько же быстры. Это немалый сюрприз для людей, выросших в убеждении, что такие операции обходятся намного дороже, чем целочисленные.

Другие базовые операции также выполняются довольно быстро. В их числе — вызовы функций, последние три строки следующей группы:

```
Integer Vector Operations
  v[i] = i                                  49
  v[v[i]] = i                               81
  v[v[v[i]]] = i                           100

Control Structures
  if (i == 5) i1++                           4
  if (i != 5) i1++                          12
  while (i < 0) i1++                         3
  i1 = sum1(i2)                             57
  i1 = sum2(i2, i3)                         58
  i1 = sum3(i2, i3, i4)                     54
```

А вот операции ввода-вывода не столь дешевы, как и другие библиотечные функции:

```
Input/Output
  fputs(s, fp)                             270
  fgets(s, 9, fp)                          222
  fprintf(fp, "%d\n", i)                  1820
  fscanf(fp, "%d", &i1)                   2070

Malloc
  free(malloc(8))                          342

String Functions
  strcpy(s, "0123456789")                  157
  i1 = strcmp(s, s)                        176
  i1 = strcmp(s, "a123456789")              64

String/Number Conversions
  i1 = atoi("12345")                       402
  sscanf("12345", "%d", &i1)              2376
  sprintf(s, "%d", i)                     1492
  f1 = atof("123.45")                     4098
  sscanf("123.45", "%f", &f1)             6438
  sprintf(s, "%6.2f", 123.45)             3902
```

Измерения для `malloc` и `free`, по всей видимости, не показывают их истинного быстродействия, поскольку освобождение памяти сразу после ее выделения — нетипичная операция.

Наконец, вот результаты для математических функций:

```
Math Functions
  i1 = rand()                                      135
  f1 = log(f2)                                     418
  f1 = exp(f2)                                     462
  f1 = sin(f2)                                     514
  f1 = sqrt(f2)                                    112
```

На другой аппаратной платформе эти значения, конечно, были бы другими. Однако эти цифры можно использовать для прикидочных расчетов времени выполнения отдельных фрагментов программ, для сравнения относительных затрат на ввод-вывод и на основные вычислительные операции или для принятия решения о том, переписывать ли выражение в другом виде или использовать встраиваемую функцию.

Данные оценки варьируются по самым разным причинам. Одна из них — это уровень оптимизации, устанавливаемый при компиляции. Современные компиляторы могут проделывать такие оптимизации, о которых программисты даже не догадываются. Более того, современные процессоры устроены так сложно, что только хорошие компиляторы могут в полной мере воспользоваться их возможностями: выполнять несколько инструкций одновременно, конвейеризировать их выполнение, доставлять инструкции и данные в регистры или кэш-память до их выполнения и т.п.

Еще одна причина, по которой трудно заранее предсказать быстродействие, — это сама архитектура вычислительных систем. Качество кэш-памяти очень сильно влияет на быстродействие; проектировщики компьютеров напрягают все силы, чтобы как-то справиться с тем обстоятельством, что оперативная память намного медленнее кэш-памяти. Сама по себе тактовая частота процессора (например, 400 МГц) хотя и говорит о быстродействии кое-что, но далеко не все. Один из наших старых компьютеров Pentium с частотой 200 МГц работает гораздо медленнее еще более старого Pentium-100 только потому, что у второго имеется большая кэш-память второго уровня, тогда как у первого ее нет. К тому же различные поколения процессоров используют разное количество тактов для выполнения тех или иных операций, даже если наборы инструкций в них совпадают.

Упражнение 7.6. Разработайте набор тестов для оценки стоимости базовых операций в тех компьютерах и компиляторах, которые вам доступны. Исследуйте сходство и различие в их быстродействии.

Упражнение 7.7. Разработайте модель стоимости средств высокого уровня в C++. Среди таких средств — создание объектов классов с помощью конструктора, их копирование и удаление; вызовы функций-методов; виртуальные функции; встраиваемые `inline`-функции; библиотека `iostream`; библиотека STL. Эта задача, в принципе, не имеет конца, поэтому сосредоточьтесь на небольшом наборе представительных операций.

Упражнение 7.8. Повторите предыдущее упражнение для языка Java.

7.7. Резюме

Если для программы выбран правильный, соответствующий задаче алгоритм, то оптимизация быстродействия — это обычно последнее, о чем следует беспокоиться при написании программы. Но если уж приходится ею заниматься, то общая процедура выглядит примерно так: провести измерения, сосредоточиться на нескольких местах, где оптимизация будет иметь наибольший эффект, проверить корректность внесенных изменений, а затем снова провести измерения. Не увлекайтесь оптимизацией; остановитесь так быстро, как это возможно. Сохраните простейшую версию как эталон для измерения времени и верификации.

При попытке оптимизировать быстродействие программы или потребления ею памяти полезно выполнять эталонные тесты и задачи, оценивая эффективность работы программы со своей точки зрения. Если для вашей задачи уже существуют стандартные тесты, воспользуйтесь ими. Если программа сравнительно автономна в своей работе, можно найти или разработать серию "типичных" наборов входных данных. Их можно включить и в серию рабочих, не эталонных тестов. Именно так организуется процесс эталонного тестирования коммерческих и научно-исследовательских систем типа компиляторов, расчетных программ и т.п. Например, вместе с Awk поставляется примерно 20 небольших программ, которые в совокупности охватывают все часто используемые средства языка. Эти программы выполняются над очень большим файлом входных данных, чтобы проверить эквивалентность получаемых результатов и отсутствие проблем с быстродействием. У нас также имеется коллекция стандартных больших файлов данных, которые можно использовать для тестов по измерению времени. В некоторых случаях полезно сделать так, чтобы эти файлы имели легко контролируемые свойства, например, длину, кратную десяти или двум.

Эталонные тесты можно проводить с помощью таких же оболочек, как те, которые рекомендованы для тестирования в главе 6, или аналогичных им. Тесты по измерению времени должны выполняться автоматически; в выходных данных должно содержаться достаточно информации для их полного понимания и воспроизведения; следует также вести регистрационные записи для отслеживания эволюционных тенденций и критическихх изменений.

Кстати, хорошие эталонные испытания провести очень трудно. Бывают фирмы, настраивающие свои программные продукты исключительно под эталонные тесты. Поэтому опубликованные результаты любых таких тестов следует воспринимать критически.

Дополнительная литература

Рассмотренный нами спам-фильтр основан на работе Боба Фландрены (Bob Flandrena) и Кена Томпсона (Ken Thompson). Их алгоритм фильтрации включает в себя регулярные выражения для более сложного поиска соответствий, а также автоматическую классификацию сообщений (гарантированный спам, вероятный спам, не спам) в соответствии с обнаруженными строками.

Статья Д. Кнута (D. Knuth) по профилированию под названием "An Empirical Study of FORTRAN Programs" была опубликована в журнале *Software — Practice and Experience*, 1971, 1, 2, p. 105–133. Основное содержание статьи составляет статистический анализ программ, обнаруженных в мусорных корзинах и общедоступных каталогах вычислительных центров.

В книгах Дж. Бентли (Jon Bentley) *Programming Pearls* и *More Programming Pearls* (Addison-Wesley, 1986 и 1988 гг. соответственно) приведены прекрасные примеры алгоритмической и настроечной оптимизации, а также имеются полезные очерки методов организации работ по оптимизации программ и профилированию.

Книга Rick Booth, *Inner Loops* (Addison-Wesley, 1997) содержит справочный материал по тонкой настройке программ для IBM-совместимых компьютеров, хотя процессоры эволюционируют так быстро, что приведенные там технические подробности стремительно устаревают.

Серия книг Джона Хеннесси (John Hennessy) и Дэвида Патерсона (David Patterson) по архитектуре компьютеров (например, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufman, 1997) содержит подробное обсуждение проблем быстродействия современной вычислительной техники.

8

Переносимость

Наконец, стандартизация — как и взаимное соглашение — является еще одним проявлением сильной упорядоченности. Но, в отличие от соглашения, современная архитектура впитала стандартизацию как явление, обогащающее саму ее технологию, хотя существуют и немалые опасения из-за доминирующего и безжалостного характера стандартизации.

Роберт Вентури. "Сложности и противоречия в архитектуре"
(Robert Venturi, Complexity and Contradiction in Architecture*)*

Разработка программы, которая бы работала правильно и эффективно, дается нелегко. Поэтому как только программа успешно заработала в одной среде, автору обычно не хочется прилагать больших усилий при переносе ее на другой процессор, компилятор или операционную систему. В идеальном случае программа вообще не должна нуждаться ни в каких изменениях.

Этот идеальный случай называется переносимостью. На практике термин "переносимость" обычно подразумевает более мягкие требования: при переносе в новую среду программу должно быть легче модифицировать, чем переписать заново. Чем меньше изменений требуется, тем лучше переносимой является программа.

Читатель может удивиться, зачем вообще беспокоиться о переносимости. Если программа должна работать только в одной рабочей среде и в заранее определенных условиях, то чего ради тратить время на расширение ее области применения? Попытаемся ответить на этот вопрос. Во-первых, всякая успешная программа неизбежно может использоваться в самых неожиданных местах и с самыми непредсказуемыми целями. Разработка программы более общей, чем это определено в ее техническом задании, позволяет меньше дорабатывать ее со временем и дает многочисленные удобства в использовании. Во-вторых, операционные среды меняются. Когда компилятор, операционная система или аппаратные компоненты подвергаются модернизации, набор рабочих средств системы изменяется. Чем меньше программа зависит от специфических возможностей системы, тем менее вероятен ее крах в новом окружении и тем проще ее адаптация к нему. Наконец, что важнее всего, хорошо переносимая программа — это всегда более качественная программа, чем плохо переносимая. Усилия, затраченные на улучшение переносимости, обычно приводят также к усовершенствованию архитектуры и реализации, а также к более тщательному тестированию. Техника программирования хорошо переносимых систем очень близка по своим принципам к хорошему стилю программирования в целом.

Разумеется, степень переносимости следует выбирать из практических соображений. Такого явления, как абсолютно переносимая программа, нет в природе; есть только программы, не протестированные в достаточно большом количестве разных сред. Но мы все же примем такое определение переносимости, как возможность выполнения программы без изменений практически в любой операционной среде. Если эта цель и не всегда достигается полностью, все же время, потраченное на улучшение переносимости при написании программы, обязательно окупится позже — хотя бы при ее доработке.

Наша основная мысль такова: постарайтесь писать программу так, чтобы она работала на пересечении нескольких различных стандартов, интерфейсов и сред, к которым она должна будет приспособиться. Не старайтесь устранить все проблемы переносимости добавлением специального кода. Вместо этого адаптируйте программу к работе в условиях новых ограничений. Используйте абстрагирование и инкапсуляцию для заключения неизбежно непереносимого кода в определенные рамки с целью более удобного управления им. Если код останется в пределах пересечения всех требований, а его зависимость от системы будет локализована, то программа станет более корректной и общей, а также потребует минимальных усилий при переносе.

8.1. Язык

Придерживайтесь стандарта. Первый шаг к хорошей переносимости кода — это, конечно, писать на языке высокого уровня, причем строго придерживаться стандарта, если таковой существует. Двоичные файлы редко являются переносимыми, а вот исходный код — вполне. Но и в этом случае способ трансляции программы в машинные коды, выполняемой компилятором, не определен строго даже для стандартизированных языков. В мире практически нет широко распространенных языков с одной-единственной реализацией; обычно среды программирования или компиляторы создаются и продаются несколькими производителями для нескольких различных операционных систем, к тому же со временем выходят все новые версии. При этом способы интерпретирования исходного кода могут эволюционировать.

Почему же стандарт не является строгим определением? Иногда стандарт просто неполон и неспособен дать точные определения в сложных случаях взаимодействия разных средств языка. Иногда стандарт намеренно сделан нестрогим. Так, тип `char` в C и C++ может иметь знак или быть беззнаковым и даже не обязан состоять ровно из 8 бит. Отдавая все эти вопросы на откуп авторам компиляторов, часто можно получить более эффективные реализации языка и избегнуть наложения излишних ограничений на аппаратные компоненты. Правда, жизнь программистов при этом усложняется. Политика тех или иных заинтересованных сторон и проблемы технической совместимости могут приводить к компромиссам, в которых подробности остаются неопределенными. Наконец, языки устроены сложно, и их компиляторы также; поэтому случаются как недочеты в интерпретации, так и ошибки в реализации.

Иногда языки бывают вообще не стандартизированы. Так, у языка C есть официальный стандарт ANSI/ISO, утвержденный в 1988 г., однако стандарт ISO для языка C++ ратифицирован только в 1998 г. Во время написания этой книги далеко не все

используемые компиляторы поддерживали официально утвержденную версию. Язык Java совсем новый, и до его стандартизации еще должны пройти годы. Стандарт языка обычно разрабатывается только после того, как появился целый ряд конфликтующих между собой реализаций, требующих согласования; при этом язык должен быть достаточно широко распространен, чтобы затраты на стандартизацию окупились. А пока суд да дело, программы писать все равно надо, и от их приспособления к различным средам тоже никуда не деться.

Итак, хотя справочники и нормативные документы создают у читателя ощущение строгой и однозначной спецификации, все же полного определения языка они никогда не дают, поэтому в разных реализациях обычно случаются различные, причем несовместимые, интерпретации стандарта. Иногда попадаются даже ошибки. Один из примеров как раз попался под руку, когда мы только писали эту главу. Вот эта внешняя декларация является недопустимой в языках С и С++:

```
?     *x[] = {"abc"};
```

Мы протестировали это выражение на добром десятке компиляторов. В результате лишь некоторые корректно диагностировали отсутствие спецификации типа `char` для переменной `x`; довольно многие выдали предупреждение о несоответствии типов (очевидно, используя старое определение языка, в котором переменная `x` по умолчанию считалась массивом указателей типа `int`); наконец, два или три скомпилировали этот некорректный код без единой жалобы.

Программируйте базовыми средствами языка. К сожалению, некоторые компиляторы не способны проследить за выполнением этого требования, представляющим собой важный аспект хорошей переносимости. В языках программирования есть “темные углы”, где полно неясностей и неоднозначностей, — например, битовые поля в С и С++. Таких мест лучше избегать. Используйте только те средства, для которых дано совершенно однозначное и хорошо понятное определение. Такие средства, по всей вероятности, общедоступны и ведут себя одинаково во всех средах. Именно их мы будем называть базовыми средствами языка.

Определить все базовые средства сразу не так-то просто; гораздо легче выявить конструкции, наверняка не входящие в их число. Совершенно новые средства, например комментарии `//` и ключевое слово `complex` в С++, или средства, специфические для одной архитектуры, наподобие ключевых слов `near` и `far`, практически наверняка вызовут проблемы. Если какое-либо средство языка настолько необычно или расплывчато определено, что для его понимания приходится обращаться к консультанту по стандартам, не пользуйтесь им ни в коем случае.

Здесь мы ограничим наше изложение языками С и С++ — распространенными языками общего назначения, широко используемыми для написания хорошо переносимых программ. Стандарт языка С имеет возраст более десяти лет, да и сам язык давно устоялся. Правда, новый стандарт уже находится в работе[1], так что следует готовиться к новым потрясениям. А вот стандарт С++ совсем свежий[2], поэтому нужно некоторое время, чтобы привести к нему все реализации языка.

---

[1] Новый стандарт ISO/ANSI С, о котором идет речь, был принят в 1999 г. Книга написана еще до этого события. — *Прим. ред.*

[2] Принят в 1998 г. — *Прим. ред.*

Что же включают в себя базовые средства C? Обычно так говорят лишь о широко распространенном и установившемся стиле программирования на этом языке, однако иногда бывает полезно подумать и о будущем. Например, в исходной версии C не требовалось объявлять прототипы функций. Функция `sqrt` объявлялась следующей декларацией:

```
?     double sqrt();
```

Здесь определен тип возвращаемого значения, но ничего не говорится о параметрах. В ANSI C были добавлены прототипы функций, в которых следовало указывать все данные:

```
double sqrt(double);
```

Компиляторы ANSI C обязаны воспринимать и старый синтаксис, но тем не менее лучше писать прототипы для всех функций. Это делает код более безопасным и устойчивым, поскольку вызовы функций полностью проверяются на совместимость типов, и если их интерфейсы меняются, компилятор это замечает. Допустим, в коде выполняется следующий вызов:

```
func(7, PI);
```

Пусть при этом функция `func` не имеет прототипа; тогда компилятор не сможет проверить, вызывается ли `func` корректно. Если впоследствии библиотека изменится и `func` будет иметь уже три аргумента, то можно упустить из виду необходимость исправить ее вызов, поскольку старый синтаксис не дает возможности проверить соответствие аргументов функций.

Язык C++ намного обширнее, чем C, и стандарт его гораздо новее, поэтому его набор базовых средств определить еще труднее. Например, хотя можно ожидать, что библиотека STL войдет в число таких средств, это произойдет еще не скоро, и некоторые нынешние реализации языка до сих пор ее не поддерживают.

Избегайте проблематичных возможностей языка. Как уже упоминалось, в стандартах некоторые моменты намеренно оставлены неопределенными или недоопределенными, обычно для того, чтобы дать разработчикам компиляторов больше свободы. Список таких моментов неприятно поражает своей длиной.

Размеры типов данных. Размеры элементарных типов данных в C и C++ не определены. Имеются только следующие базовые правила:

```
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double)
```

Тип `char` должен иметь длину как минимум 8 бит, `short` и `int` — как минимум 16, а `long` — не менее 32. Никакие другие свойства не гарантируются. Отсутствует даже требование, чтобы значение указателя помещалось в переменную типа `int`.

Для конкретного компилятора можно довольно просто выяснить, каковы именно размеры типов данных:

```
/* sizeof: отображает длины элементарных типов */
int main(void)
{
    printf("char %d, short %d, int %d, long %d,",
```

```
        sizeof(char), sizeof(short),
        sizeof(int), sizeof(long));
    printf(" float %d, double %d, void* %d\n",
        sizeof(float), sizeof(double), sizeof(void *));
    return 0;
}
```

Эта программа выдает одни и те же результаты в большинстве систем, которыми мы регулярно пользуемся:

```
char 1, short 2, int 4, long 4, float 4, double 8, void* 4
```

Но возможны и другие значения. Некоторые 64-разрядные системы выдают такой результат:

```
char 1, short 2, int 4, long 8, float 4, double 8, void* 8
```

Старые компиляторы для персональных компьютеров IBM PC обычно давали следующий ответ:

```
char 1, short 2, int 2, long 4, float 4, double 8, void* 2
```

В ранние годы существования персональных компьютеров их аппаратура поддерживала несколько видов указателей. Чтобы справиться с этой неразберихой, были придуманы модификаторы `near` и `far`, ни один из которых не является стандартным. Тем не менее эти призраки прошлого все еще обитают в нынешних компиляторах, поскольку являются зарезервированными ключевыми словами. Если ваш компилятор может изменять размеры элементарных типов, или если в вашем распоряжении есть системы с разными размерами, попытайтесь скомпилировать и протестировать вашу программу в этих различных конфигурациях.

В стандартном заголовочном файле `stddef.h` определен ряд типов, помогающих улучшить переносимость. Наиболее часто используемым из них является `size_t` — целочисленный тип без знака, возвращаемый операцией `sizeof`. Также значения этого типа возвращаются функциями наподобие `strlen` и служат аргументами многих других функций, включая `malloc`.

Поучившись на ошибках других, авторы Java четко определили размеры всех базовых типов данных: 8 бит для `byte`, 16 — для `char` и `short`, 32 — для `int` и 64 — для `long`.

Здесь мы не будем обсуждать богатый набор потенциальных проблем, связанных с вещественнозначными операциями с плавающей точкой, поскольку эта тема могла бы занять отдельную книгу. К счастью, большинство современных систем поддерживает стандарт IEEE для аппаратных компонентов, предназначенных для вещественных операций, так что характеристики вещественнозначной арифметики определены довольно неплохо.

Порядок вычисления выражений. В языках C и C++ порядок вычисления операндов выражений и аргументов функций, а также некоторых других значений не определен. Рассмотрим следующее присваивание:

```
?    n = (getchar() << 8) | getchar();
```

Вторая функция `getchar` могла быть вызвана первой, поскольку порядок записи выражения не всегда совпадает с порядком его выполнения. Еще один пример:

```
?    ptr[count] = name[++count];
```

Здесь переменная `count` может вычисляться до или после ее использования в качестве индекса массива `ptr`. Рассмотрим следующий оператор для немедленного вывода введенных символов:

```
?    printf("%c %c\n", getchar(), getchar());
```

В нем первый введенный символ вполне может выводиться вторым, а не первым. А вот оператор с намеренно сделанной ошибкой выполнения:

```
?    printf("%f %s\n", log(-1.23), strerror(errno));
```

Здесь значение `errno` может вычисляться еще до вызова функции `log`.

Выполнение некоторых выражений подчиняется определенным правилам. По определению все побочные эффекты и вызовы функций должны завершаться к моменту, когда в программе встретится очередная точка с запятой, или к следующему вызову функции. Операции `&&` и `||` выполняются слева направо, причем только в той степени, в которой это необходимо для вычисления всего выражения (включая и побочные эффекты). В операторе `? :` вычисляется условие (учитывая сторонние эффекты) и только затем вычисляется строго одно из следующих после точки с запятой выражений.

В языке Java порядок вычислений определен значительно строже. В нем требуется, чтобы все выражения, включая побочные эффекты, вычислялись слева направо. Тем не менее один авторитетный справочник советует не полагаться на это правило в ответственных случаях. Это вполне здравый совет, если есть хоть какая-то вероятность, что код будет переводиться на язык C или C++, где подобных гарантий нет. Вообще, перевод с одного языка на другой — это экстремальный, но иногда полезный способ протестировать переносимость алгоритма.

Наличие знака у типа `char`. В языках C и C++ не определено, должен ли тип `char` быть знаковым или беззнаковым. От этого могут возникнуть проблемы — в частности, при сочетании типов `char` и `int`, как это происходит при вызове функции `getchar()` с целым возвращаемым значением. Рассмотрим следующий код:

```
?    char c;   /* должен иметь тип int */
?    c = getchar();
```

В результате значение `c` будет находиться в диапазоне от `0` до `255`, если тип `char` не имеет знака, и от `-128` до `127` в противном случае. Здесь имеется в виду почти универсальная конфигурация 8-битовых символов в системе с дополнением к двум. В этом случае возникают неприятные последствия, если символ должен использоваться в качестве индекса массива или сравниваться с кодом `EOF`, который обычно определен как `-1` в файле `stdio.h`. Например, для раздела 6.1 мы разработали приведенный ниже код, после того как в исходной версии пришлось исправить несколько предельных случаев. Так, выражение `s[i] == EOF` никогда не бывает истинным, если тип `char` не имеет знака:

```
?    int i;
?    char s[MAX];
?
?    for (i = 0; i < MAX-1; i++)
?        if ((s[i] = getchar()) == '\n' || s[i] == EOF)
?            break;
?    s[i] = '\0';
```

Если функция getchar возвратит EOF, в s[i] будет помещено значение 255 (0xFF, т.е. результат преобразования -1 к типу unsigned char). Если элемент s[i] имеет тип без знака, его значение так и останется 255, и при сравнении с EOF это даст ложный результат.

Но даже если тип char имеет знак, приведенный код все равно ошибочен. Да, в этом случае сравнение с EOF даст нужный результат, однако вполне корректный входной байт 0xFF не будет отличаться от EOF и может привести к преждевременному завершению цикла. Итак, независимо от наличия знака у переменных типа char, необходимо всегда помещать возвращаемое из getchar значение в переменную типа int для сравнения с EOF. Вот как нужно записать этот цикл в переносимой форме:

```
int c, i;
char s[MAX];
for (i = 0; i < MAX-1; i++) {
    if ((c = getchar()) == '\n' || c == EOF)
        break;
    s[i] = c;
}
s[i] = '\0';
```

В языке Java нет модификатора unsigned; целочисленные типы в нем имеют знак, а 16-разрядный тип char — не имеет.

Арифметический или логический сдвиг. Сдвиг чисел со знаком вправо с помощью операции >> может быть арифметическим (копия знакового бита размножается в процессе сдвига) или логическим (освобождаемые при сдвиге биты замещаются нулями). И снова, научившись на горьком опыте C и C++, создатели Java зарезервировали знак >> для арифметического сдвига вправо, а для логического предусмотрели отдельный знак — >>>.

Порядок байтов. Порядок следования байтов в типах short, int и long не определен; байт с самым низким адресом может оказаться как младшим, так и старшим значащим байтом. Это зависит от аппаратуры компьютера. Мы обсудим этот вопрос более подробно чуть позже в этой же главе.

Выравнивание членов структур и классов. Выравнивание составных элементов внутри структур, классов и объединений не определено полностью; единственное, что сказано по этому поводу, — элементы должны следовать в порядке их объявления. Рассмотрим следующую структуру:

```
struct X {
    char c;
    int i;
};
```

В ней элемент i может иметь смещение 2, 4 или 8 байт от начала структуры. Некоторые системы позволяют размещать переменные типа int по нечетным адресам, но большинство требует, чтобы элемент типа, состоящего из n байт, помещался по адресу, кратному n. Например, значения типа double, обычно занимающие 8 байт, помещаются по адресам, кратным восьми. Кроме всего этого, автор компилятора может добавить свои собственные модификации, например, вынужденное выравнивание по соображениям быстродействия.

Никогда не следует предполагать, что элементы структуры занимают непрерывную область памяти. Выравнивание создает "дыры", поэтому, например, структура X будет содержать как минимум один байт лишней, неиспользуемой памяти. Наличие таких "дыр" подразумевает, что структура в целом может занимать больше места, чем сумма длин ее элементов, причем ее длина может варьироваться от одной системы к другой. При динамическом распределении памяти для такой структуры следует запрашивать sizeof(struct X) байт, а не sizeof(char) + sizeof(int).

Битовые поля. Битовые поля настолько зависят от аппаратуры и системы, что для обеспечения переносимости ими вообще не нужно пользоваться.

Этот долгий список смертельных опасностей можно укоротить, следуя нескольким правилам. Не пользуйтесь побочными эффектами в выражениях, кроме очень немногих идиоматических конструкций:

```
a[i++] = 0;
c = *p++;
*s++ = *t++;
```

Не сравнивайте значение типа char с кодом EOF. Для вычисления длины типа или объекта пользуйтесь операцией sizeof. Никогда не выполняйте сдвиг вправо над значением со знаком. Позаботьтесь о достаточной длине типа данных для любых возможных значений, которые могут присваиваться соответствующей переменной.

Испытайте несколько компиляторов. Можно вообразить себе, что вам уже понятны все вопросы переносимости, однако компиляторы способны обнаружить такие проблемы, которых вы никогда не увидите. Разные компиляторы иногда воспринимают одну и ту же программу по-разному, поэтому следует пользоваться их помощью. Включите все предупреждения. Испытайте несколько компиляторов в одной и той же системе, а также в разных системах. Скомпилируйте программу на языке C компилятором C++.

Поскольку один и тот же язык может отличаться в средах различных компиляторов, тот факт, что программа компилируется без ошибок одним компилятором, еще не гарантирует даже ее синтаксической правильности. Если же несколько компиляторов принимает ваш код без ошибок, ситуация улучшается. Мы компилировали все программы на C из этой книги тремя разными компиляторами в трех разных и не родственных операционных системах (Unix, Plan 9, Windows), а также одним-двумя компиляторами C++. Это был весьма отрезвляющий опыт, поскольку все эти компиляторы выловили полтора десятка ошибок переносимости, найти которые вручную было бы не в силах человеческих. А вот исправление этих ошибок потребовало вполне тривиальных усилий.

Разумеется, компиляторы могут и сами порождать проблемы совместимости, выбирая различные варианты нестрого заданных операций. Но наш подход все же обнадеживает. Чем писать код таким образом, что различия между системами, средами и компиляторами только подчеркиваются и усиливаются, мы пытаемся организовать программу так, чтобы она не зависела от среды. Короче говоря, мы попросту пытаемся избежать средств и возможностей, подверженных вариациям от системы к системе.

8.2. Заголовочные файлы и библиотеки

Заголовочные файлы и библиотеки предоставляют программисту дополнительные средства для работы, помимо тех, которые обеспечиваются основным синтаксисом языка. Среди характерных примеров — средства ввода-вывода `stdio` в C, `iostream` в C++, или `java.io` в Java. Строго говоря, эти средства не являются частью самих языков, но определены вместе с ним и должны составлять неотъемлемую часть любой среды, в которой декларируется поддержка этих языков. Но поскольку эти библиотеки охватывают слишком широкий спектр операций и часто должны близко взаимодействовать с операционной системой, в них по-прежнему скрываются проблемы непереносимости.

Пользуйтесь стандартными библиотеками. В отношении библиотек применим тот же совет, что и для основной части языка: придерживайтесь стандарта, причем его старых, отработанных компонентов. В языке C определяются стандартные библиотеки функций для ввода-вывода, строковых операций, анализа классов символов, распределения памяти и ряда других задач. Если ограничить взаимодействие программы с операционной системой этими функциями, то вероятность стабильного, одинакового выполнения кода в разных системах серьезно повышается. Тем не менее нужно быть осторожным, потому что существует много реализаций библиотек и некоторые из них содержат нестандартные возможности.

В ANSI C не определена функция копирования строк `strdup`, однако в большинстве сред она существует даже при том, что эти среды декларируют свое полное соответствие стандарту. Опытный программист может воспользоваться функцией `strdup` по привычке, и среда разработки даже не предупредит его о нестандартности данного средства. Позже программа не пройдет компиляцию при ее переносе в среду, не поддерживающую данную функцию. Именно такого типа проблемы переносимости обычно связаны с библиотеками. Единственное возможное решение — это строго придерживаться стандарта и тестировать программу в самых разнообразных рабочих средах.

Заголовочные файлы и определения пакетов задают интерфейс к стандартным функциям. Существует проблема, связанная с загромождением таких файлов, поскольку их авторы стараются удовлетворить требованиям сразу нескольких языков. Например, можно обнаружить один файл `stdio.h`, обслуживающий одновременно компиляторы старого C, ANSI C и даже C++. В таких случаях файл изобилует директивами условной компиляции `#if` и `#ifdef`. Язык директив препроцессора не отличается гибкостью, поэтому такие файлы трудно читать и понимать, и они иногда содержат ошибки.

Следующий отрывок из заголовочного файла одной из наших систем лучше других, хотя бы потому, что он хорошо отформатирован:

```
?    #ifdef _OLD_C
?       extern int fread();
?       extern int fwrite();
?    #else
?    # if defined(__STDC__) || defined(__cplusplus)
?         extern size_t fread(void*, size_t, size_t, FILE*);
?         extern size_t fwrite(const void*, size_t, size_t, FILE*);
?    # else /* not __STDC__ || __cplusplus */
?         extern size_t fread();
?         extern size_t fwrite();
?    # endif /* else not __STDC__ || __cplusplus */
?    #endif
```

Этот пример не содержит ошибок, но он демонстрирует, что подобные заголовочные файлы (и программы) устроены слишком сложно и с трудом поддаются доработке. Было бы легче использовать отдельный файл для каждого компилятора или среды. Конечно, пришлось бы работать сразу с множеством таких файлов (обеспечивая их поддержку и доработку), но каждый из них был бы самодостаточным и полностью приспособленным к своей системе. Это уменьшило бы вероятность таких ошибок, как включение функции strdup в среду строгого стандарта ANSI С.

Заголовочные файлы также могут "загрязнить" пространство имен, объявив функцию с тем же именем, что и в программе. Например, наша функция для выдачи предупреждающих сообщений, weprintf, ранее называлась wprintf, но потом мы обнаружили, что в некоторых средах функция с таким именем уже определена в заголовочном файле stdio.h в ожидании нового стандарта ANSI С. Поэтому нам пришлось изменить имя функции, чтобы программа компилировалась без проблем во всех средах и в будущем ее не нужно было бы изменять. Если бы проблема состояла в ошибочной реализации, а не в нормальном изменении спецификации, мы бы могли обойти эту трудность, переопределив имя при включении заголовочного файла:

```
?    /* некоторые версии stdio используют wprintf, поэтому заменяем
имя: */
?    #define wprintf stdio_wprintf
?    #include <stdio.h>
?    #undef wprintf
?    /* далее код, использующий wprintf()... */
```

Этот код преобразует все вхождения wprintf в заголовочном файле в вызовы stdio_wprintf, чтобы они не пересекались с нашей версией. Теперь мы можем пользоваться нашей функцией wprintf, не изменяя ее имени. Цена, которую мы за это платим, — некоторая неуклюжесть кода и риск того, что компонуемая с нашей программой библиотека вызовет функцию wprintf, ожидая стандартную, а получая нашу. В отношении одной функции, по-видимому, особенно волноваться не приходится, но некоторые системы создают такую путаницу в среде программирования, что для сохранения хорошего стиля приходится идти на крайние меры. Не забывайте комментировать, что именно делает конструкция, и не ухудшайте ее работу добавлением условной компиляции. Если функция wprintf определяется в некоторых средах, следует предполагать, что то же самое происходит во всех. Тогда

одного исправления будет достаточно и не придется сохранять множество операторов `#ifdef`. Иногда лучше уклониться от конфликта, чем вступать в него (и это, несомненно, безопаснее). Именно так мы поступили, изменив имя функции на `weprintf`.

Даже если изо всех сил стараться придерживаться стандарта и работать в хорошо отлаженной среде, все равно бывает легко выйти за рамки, неявно предположив, что какое-то свойство имеет место всегда и везде. Например, в ANSI C определяется шесть сигналов, которые можно перехватить функцией `signal`; в стандарте POSIX таковых уже 19; в большинстве версий Unix поддерживается 32 или больше сигналов. Если вы хотите воспользоваться нестандартным сигналом, то придется идти на компромисс между функциональностью и переносимостью, решая на ходу, что важнее.

Существует много других стандартов, не являющихся частью определения языка. Среди них операционные системы, сетевые интерфейсы, графические интерфейсы и т.д. Некоторые работают в нескольких системах, как POSIX; другие специфичны для одной системы, как многочисленные API-интерфейсы Microsoft Windows. Здесь также применим все тот же совет. Ваши программы будут лучше переносимы, если выбрать широко распространенные, хорошо определенные стандарты и придерживаться только самых главных, общепринятых аспектов этих стандартов.

8.3. Организация программы

Существуют два основных подхода к переносимости, которые мы называем "объединением" и "пересечением". "Объединение" подразумевает использование лучших черт каждой конкретной системы и соответственно организацию условной компиляции и инсталляции в зависимости от свойств среды. Получившийся код реализует объединение всех возможных сценариев и пользуется преимуществами всех систем. Недостатки этого подхода состоят в размере кода и сложности процесса установки программы, а также в запутанности исходного текста ввиду наличия большого количества операторов условной компиляции.

Пользуйтесь только средствами, имеющимися в наличии везде. Мы рекомендуем именно подход "пересечения": пользоваться только теми чертами и возможностями языка, которые существуют во всех системах, куда может переноситься код, и не пользоваться теми, которые имеются не везде. Опасность этого подхода состоит в том, что требование общедоступности возможностей может сузить круг систем, в которые может переноситься программа; также имеется риск падения быстродействия.

Чтобы сравнить эти два подхода, рассмотрим несколько примеров, в которых используется "объединение", и перепроектируем их под "пересечение". Как мы покажем далее, код с "объединением" непереносим по самой своей структуре, несмотря на декларируемые этим подходом цели, тогда как код с "пересечением" не только переносим, но и устроен гораздо проще.

Следующий небольшой пример представляет попытку приспособиться к среде, которая по какой-то причине не содержит стандартного заголовочного файла `stdlib.h`:

```
?    #if defined (STDC_HEADERS) || defined (_LIBC)
?    #include <stdlib.h>
?    #else
?    extern void *malloc(unsigned int);
?    extern void *realloc(void *, unsigned int);
?    #endif
```

Эта мера предосторожности вполне приемлема, если используется время от времени, но не слишком часто. В связи с ней напрашивается также вопрос, сколько еще функций из `stdlib` придется декларировать в этом или аналогичном условно компилируемом коде. Например, если используются функции `malloc` и `realloc`, то понадобится также и `free`. А что если тип `unsigned int` отличается по размеру от `size_t`, т.е. корректного типа аргументов `malloc` и `realloc`? Более того, как нам узнать, определены ли константы `STDC_HEADERS` или `_LIBC`, причем определены ли они корректно? Откуда нам знать, не существует ли другого имени, которое может привести к конфликту или подмене имен в какой-либо системной среде? Любой код с условной компиляцией — наподобие этого — неполон в плане переносимости, потому что рано или поздно встретится система, не вписывающаяся в заранее определенные рамки, и операторы `#ifdef` придется редактировать. Если бы удалось вовсе устранить условную компиляцию, мы бы решили проблему с последующей доработкой кода.

Задача, которую решает этот код, вполне реальна. Так как же нам решить ее раз и навсегда? Мы бы предпочли предполагать, что существуют стандартные заголовочные файлы; если это не так, то это проблема других людей. Но при отсутствии таких файлов будет проще включить в комплект поставки программы специальный заголовочный файл, в котором функции `malloc`, `realloc` и `free` определяются точно так же, как в ANSI C. Этот файл можно включать в программу при любых условиях, вместо того чтобы лепить на код "заплатки" то тут, то там. Таким способом мы гарантируем, что необходимый интерфейс всегда будет в наличии.

Избегайте условной компиляции. Условная компиляция с помощью операторов `#ifdef` и других подобных директив препроцессора трудна в управлении, поскольку таким образом информация рассеивается по всему исходному коду.

```
?    #ifdef NATIVE
?        char *astring = "convert ASCII to native character set";
?    #else
?    #ifdef MAC
?        char *astring = "convert to Mac textfile format";
?    #else
?    #ifdef DOS
?        char *astring = "convert to DOS textfile format";
?    #else
?        char *astring = "convert to Unix textfile format";
?    #endif /* ?DOS */
?    #endif /* ?MAC */
?    #endif /* ?NATIVE */
```

Этот отрывок лучше было бы структурировать с помощью `#elif` после каждого определения, вместо того чтобы нагромождать `#endif` в самом конце. Однако настоящая проблема состоит в том, что, несмотря на намерение авторов, этот код исключительно плохо переносим, поскольку ведет себя по-разному в разных систе-

мах и требует добавления нового #ifdef для каждой новой среды. Было бы проще ввести одну строку с более общим текстом, сделав код более простым, полностью переносимым и столь же информативным:

```
char *astring = "convert to local text format";
```

Здесь условная компиляция уже не нужна, поскольку строка останется одной и той же во всех системах.

Смешение условной компиляции (выполняемой операторами #ifdef) с контрольными операторами программы очень неудачно, поскольку такой код трудно воспринимать.

```
?   #ifndef DISKSYS
?       for (i = 1; i <= msg->dbgmsg.msg_total; i++)
?   #endif
?   #ifdef DISKSYS
?       i = dbgmsgno;
?       if (i <= msg->dbgmsg.msg_total)
?   #endif
?       {
?           ...
?           if (msg->dbgmsg.msg_total == i)
?   #ifndef DISKSYS
?               break; /* больше сообщений не ожидается */
?           еще около 30 строк, также с условной компиляцией
?   #endif
?       }
```

Даже если условная компиляция кажется безвредной, ее часто можно заменить более безопасными методами. Например, директивы #ifdef часто используются для управления отладочным кодом:

```
?   #ifdef DEBUG
?       printf(...);
?   #endif
```

Однако столь же успешно можно использовать и обычный оператор if с постоянным условием:

```
enum { DEBUG = 0 };
...
if (DEBUG) {
    printf(...);
}
```

Если константа DEBUG равна нулю, большинство компиляторов не сгенерируют никакого кода для блока в условном операторе, однако проверят синтаксис этого блока. А вот #ifdef, наоборот, скроет возможные синтаксические ошибки, которые могут проявиться впоследствии при компиляции соответствующего блока.

Иногда при условной компиляции исключаются очень большие блоки кода:

```
#ifdef notdef /* неопределенный символ */
    ...
#endif
```

Или же:

```
#if 0
    ...
#endif
```

Однако условных конструкций можно вообще избежать, вместо этого подставляя по условию различные файлы во время компиляции. К этой теме мы вернемся в следующем разделе.

Если приходится модифицировать программу под новую среду, не начинайте с создания копии всей программы. Вместо этого попытайтесь адаптировать оригинальный код. Вероятно, потребуется внести изменения в основной массив кода, и если делать это с копией, то вскоре возникнут различные версии с сильными расхождениями. Насколько это возможно, следует держать у себя только один экземпляр исходного кода. Если обнаруживается, что необходимо изменить что-нибудь для приспособления кода к конкретной среде, найдите способ внести такие изменения, чтобы они работали везде. Измените внутренние интерфейсы, если это необходимо, однако сохраните код самосогласованным и избегайте директив `#ifdef`. Это обеспечит переносимость кода в дальней перспективе, а не его узкую специализацию. Сузьте "пересечение", а не расширяйте "объединение".

Мы уже высказались против условной компиляции и продемонстрировали некоторые вызываемые ею проблемы. Однако мы еще не упомянули самую сложную из них: такой код практически невозможно тестировать. Одна директива `#ifdef` превращает одну программу в две, компилируемые отдельно. При этом трудно судить, все ли вариации одной программы были скомпилированы и протестированы. Если в один блок `#ifdef` вносится изменение, оно же может понадобиться и в других, однако проверить его работу можно только в тех средах, в которых активизируются соответствующие блоки. Если аналогичное изменение необходимо внести в другие конфигурации, его нельзя протестировать. Если мы добавляем новый блок `#ifdef`, трудно изолировать данное изменение от других и определить, каким еще условиям необходимо удовлетворить, чтобы попасть в это место программы, и где еще может понадобиться устранение этой же проблемы. Наконец, если в коде какая-то часть опускается по условию, то компилятор просто ее не воспринимает. Эта часть может быть полной чепухой, но мы этого не узнаем до тех пор, пока какой-нибудь невезучий клиент не попытается скомпилировать ее в среде, где активизируется именно эта часть. Так, следующая программа компилируется, если определена константа _MAC, и не компилируется в противном случае:

```
#ifdef _MAC
    printf("This is Macintosh\r");
#else
    Синтаксическая ошибка в других системах
#endif
```

Итак, наши предпочтения состоят в том, чтобы пользоваться только общими чертами для всех сред. В этом случае можно скомпилировать и протестировать весь код. Если где-то имеется проблема с переносимостью, мы переписываем это место, вместо того чтобы добавлять условно компилируемые блоки. Следуя этим путем, мы

непрерывно улучшаем переносимость и качество программы в целом, вместо того чтобы неоправданно усложнять ее.

Некоторые большие системы поставляются вместе со сценариями конфигурирования, позволяющими подстроить код под конкретную локальную среду. В процессе компиляции этот сценарий анализирует свойства среды: местонахождение заголовочных файлов и библиотек, порядок байтов внутри машинных слов, размеры типов данных, известные некорректные реализации языков (а таких на удивление много) и т.д. Затем сценарий генерирует параметры конфигурации или файлы связей проекта, обеспечивающие адаптацию программы к среде. Такие сценарии бывают длинными и сложными, представляют собой существенную часть дистрибутивного пакета программы и требуют постоянного контроля для поддержания работоспособности. Иногда применение подобных методов необходимо, но, вообще-то, чем более код переносим и чем меньше в нем директив `#ifdef`, тем проще и надежнее его конфигурирование и установка.

Упражнение 8.1. Исследуйте вопрос о том, как ваш компилятор воспринимает код внутри следующего условного блока:

```
const int DEBUG = 0;
/* или enum { DEBUG = 0 }; */
/* или final boolean DEBUG = false; */
if (DEBUG) {
    ...
}
```

При каких обстоятельствах здесь проверяется синтаксис? Когда генерируется код? Если в вашем распоряжении имеется несколько компиляторов, как соотносятся результаты их работы?

8.4. Изоляция

Хотя нам всегда хотелось бы иметь один пакет файлов исходного кода, которые бы компилировались без всяких изменений во всех системах, этот идеал практически недостижим. Однако ошибкой было бы и рассеивание непереносимого кода по всей программе, а именно эту проблему создает условная компиляция.

Локализуйте зависимость от системы в отдельных файлах. Если для разных систем требуется разный код, эти различия следует локализовать в отдельных файлах — по одному на каждую систему. Например, текстовый редактор Sam работает в Unix, Windows и еще нескольких операционных системах. Системные интерфейсы в этих средах отличаются очень существенно, однако большая часть кода программы Sam остается в них неизменной. Вариации, присущие конкретной среде, собраны в одном файле; так, файл `unix.c` содержит интерфейсный код для систем Unix, а `windows.c` — для систем Windows. Эти файлы реализуют интерфейс с операционной системой и скрывают различия от остальных модулей. На самом деле редактор Sam вообще написан для своей собственной виртуальной операционной системы, переносимой в различные реальные системы путем дописывания двух сотен

строк на С, которые реализуют с полдесятка небольших, но непереносимых операций с использованием местных системных вызовов.

Графические среды упомянутых операционных систем практически не имеют между собой ничего общего. Программа Sam справляется с этой проблемой, имея собственную переносимую библиотеку для своей графики. Хотя разработка такой библиотеки требует гораздо бо́льших усилий, чем подгонка кода под заданную систему (например, код интерфейса с системой X Window всего в два раза короче, чем вся остальная часть программы Sam), в дальней перспективе это оказывается выгоднее. Имеется и побочная выгода — эта графическая библиотека сама по себе представляет ценность и уже использовалась отдельно для того, чтобы сделать переносимыми несколько других программ.

Sam — довольно старая программа; в настоящее время для целого ряда платформ уже существуют переносимые графические среды, такие как OpenGL, Tcl/Tk и Java. Написание кода с использованием их, а не своих собственных разработок позволит придать вашим программам более всеобъемлющий характер.

Скрывайте зависимость от системы за интерфейсами. Для проведения четкой границы между переносимой и непереносимой частями кода очень удобно использовать механизмы абстрагирования. Хорошим примером могут служить библиотеки ввода-вывода, существующие в большинстве языков программирования: они представляют внешнюю память в виде набора файлов, которые можно открывать и закрывать, считывать и записывать, не беспокоясь об их физическом местонахождении или о структуре. Программы, придерживающиеся какого-либо подобного интерфейса, будут работать на любой платформе, который его поддерживает.

Реализация программы Sam — это еще один пример абстрагирования. Для файловой системы и графических операций определен интерфейс, и программа использует только его средства и возможности. Сам интерфейс опирается на те средства, которые имеются в конкретной операционной системе. Реализации этих интерфейсов могут очень сильно отличаться в разных системах, но программы, использующие сами интерфейсы, совершенно независимы и при переносе не требуют никаких изменений.

Подход к переносимости в языке Java показывает, как далеко можно зайти в развитии этой идеи. Программа на Java транслируется в последовательность операций "виртуальной машины", т.е. имитатора реальной системы, который легко реализовать в любой конкретной операционной среде. Библиотеки Java обеспечивают единообразный доступ к средствам операционной среды, таким как графика, пользовательский интерфейс, сеть и т.д.; средства самих библиотек опираются на то, что имеется в системах. Теоретически одну и ту же программу на Java можно выполнить в любой среде без изменений (даже после ее трансляции).

8.5. Обмен данными

Текстовые данные легко переносятся из одной системы в другую, и это самый простой способ обмена информацией произвольного содержания между системами.

Используйте текстовые форматы для обмена данными. Текст легко обрабатывать самыми разными программами и способами. Например, если выходные данные одной программы не вполне соответствуют требованиям другой, для которой

они должны служить входными данными, то для их исправления можно применить сценарий на языке Awk или Perl. Для выбора или отбрасывания строк можно воспользоваться утилитой grep; более сложные изменения могут выполняться с помощью текстового редактора. Текстовые файлы также гораздо легче документировать, а иногда документирование вообще не нужно, поскольку люди и так могут их прочитать. С помощью комментария в текстовом файле можно указать, какая версия программы нужна для его обработки. Например, первая строка PostScript-файла указывает его кодировку:

```
%!PS-Adobe-2.0
```

Напротив, двоичные файлы требуют для своей обработки строго специализированных программ и редко могут использоваться совместно даже в одной системе. Существует ряд программ, которые преобразуют любые двоичные данные в текст для их передачи с меньшей вероятностью повреждения. Среди них — binhex для систем Macintosh, uuencode и uudecode для Unix, а также различные программы MIME-кодирования для передачи двоичных данных в почтовых сообщениях. В главе 9 демонстрируется семейство функций кодирования и декодирования двоичных данных в пакеты и из пакетов для пересылки между системами. Само разнообразие подобных программ уже говорит о проблемах, связанных с двоичными форматами.

С обменом текстовыми данными связана одна старая, давно всем надоевшая проблема: в IBM PC-совместимых системах для завершения строки используется пара символов "возврат каретки" (\r) и "перевод строки" (\n), в то время как в Unix — только символ новой строки. "Возврат каретки" — это пережиток, оставшийся от допотопного устройства под названием телетайп, в котором существовала операция возврата каретки (CR) для перемещения печатающего механизма на начало строки, а также отдельная операция перевода строки (LF) для перемещения к началу следующей строки.

Даже при том, что в современных компьютерах нет никаких возвращаемых "кареток", программы для PC все равно большей частью ожидают именно эту комбинацию (общеизвестную под названием CRLF, что произносится как "керлиф") в конце каждой строки. Если возврата каретки нет, весь файл может быть воспринят как одна гигантская строка. Подсчет строк или символов может оказаться неверным или произвольно изменяться. Некоторые программы справляются с этой ситуацией корректно, однако большинство — нет. Системы для PC — не единственные виновники; из-за цепочки последовательных попыток поддержания обратной совместимости некоторые современные стандарты сетевой связи, такие как HTTP, также используют комбинацию CRLF для отделения строк друг от друга.

Наш совет — пользоваться стандартным интерфейсом, который воспринимает комбинацию CRLF последовательно и согласованно в каждой конкретной системе. Так, на PC выполняется удаление символа \r при вводе и его возврат на место при выводе. В Unix для отделения строк в файлах всегда используется символ \n, а не CRLF. Если необходимо часто переносить файлы с одних машин на другие, то просто необходимо иметь программу для автоматического преобразования текстовых форматов.

Упражнение 8.2. Напишите программу, которая бы удаляла лишние символы возврата каретки в файле. Напишите вторую программу, которая бы добавляла их путем замены всех символов конца строки на символы возврата каретки плюс такие же символы конца строки. Как протестировать эти программы?

8.6. Порядок следования байтов

Несмотря на вышеописанные недостатки, двоичные данные все-таки бывают необходимы. Они более компактны, их быстрее декодировать, и эти факторы часто делают их удобными для многих задач обработки компьютерной информации в сетях и т.п. Однако с двоичными данными связаны серьезные проблемы переносимости.

По крайней мере одну проблему все-таки удалось решить окончательно: все современные системы основаны на 8-битовых байтах. Однако в различных системах объекты длиннее одного байта уже могут представляться по-разному, поэтому полагаться на их специфические свойства нельзя. Например, в коротком целом числе (`short int`, обычно 16 бит или 2 байта) младший байт может помещаться по меньшему адресу, чем старший (так называемый “остроконечный” принцип, “little-endian”), или по большему адресу (“тупоконечный” принцип, “big-endian”). Этот выбор довольно произволен, а некоторые системы даже поддерживают обе модели.

Итак, хотя системы-“остроконечники” и системы-“тупоконечники” воспринимают память как последовательность слов в одном и том же порядке, они могут отсчитывать байты внутри слов по-разному. На приведенной ниже схеме четыре байта по адресу `0` представляют шестнадцатеричное целое число `0x11223344` на “тупоконечной” машине и `0x44332211` — на “остроконечной”.

0	1	2	3	4	5	6	7	8				
11	22	33	44									

Чтобы увидеть проблему в действии, протестируйте следующую программу:

```
/* byteorder: выводит байты числа типа long */
int main(void)
{
    unsigned long x;
    unsigned char *p;
    int i;
    /* 11 22 33 44 => тупоконечник */
    /* 44 33 22 11 => остроконечник */
    /* x = 0x1122334455667788UL; для 64-бит.long */
    x = 0x11223344UL;
    p = (unsigned char *) &x;
    for (i = 0; i < sizeof(long); i++)
        printf("%x ", *p++);
    printf("\n");
    return 0;
}
```

В 32-разрядной "тупоконечной" системе на выходе получаем:

```
11 22 33 44
```

Однако в "остроконечной" системе результат будет таким:

```
44 33 22 11
```

На машине PDP-11 (это давно устаревшая 16-разрядная ЭВМ, все еще встречающаяся в виде встроенных контроллеров) получим следующее:

```
22 11 44 33
```

В системах с 64-битовым типом `long` можно сделать константы побольше и получить аналогичный результат.

Эта программа может показаться пустым развлечением, но это не так. Если необходимо послать целое число по параллельному соединению шириной в один байт, например по сетевому кабелю, мы должны выбрать, какой байт посылать первым. Тут-то и возникает главный вопрос — выбор между "тупоконечниками" и "остроконечниками". Другими словами, программа делает явным образом примерно то же, что следующий вызов делает неявно:

```
fwrite(&x, sizeof(x), 1, stdout);
```

Рискованно записывать целое число типа `int` (а также `short` или `long`) на одном компьютере, а затем считывать его тоже как `int` на другом.

Пусть передающий компьютер записывает данные, используя такой код:

```
unsigned short x;
fwrite(&x, sizeof(x), 1, stdout);
```

Принимающий компьютер считывает их этим кодом:

```
unsigned short x;
fread(&x, sizeof(x), 1, stdin);
```

Если в двух системах порядок байтов отличается, значение `x` не сохранится. Если `x` вначале равно `0x1000`, то прибыть оно может уже как `0x0010`.

Эту проблему часто решают условной компиляцией и перестановкой байтов, т.е. примерно так:

```
?    short x;
?    fread(&x, sizeof(x), 1, stdin);
?    #ifdef BIG_ENDIAN
?    /* перестановка байтов */
?    x = ((x&0xFF) << 8) | ((x>>8) & 0xFF);
?    #endif
```

Этот подход становится совсем неудобным, если нужно обмениваться многими двух- и четырехбайтовыми числами. На практике байты приходится постоянно переставлять, поскольку они перемещаются с места на место.

Уж если положение с типом `short` настолько плохо, можете себе представить, что творится с более длинными типами данных, для которых возможно большее количество перестановок. Добавьте сюда выравнивание переменных-элементов струк-

тур, дополнение членов до слов пустыми байтами, загадочный порядок байтов на старых машинах, и проблема приобретет совершенно неразрешимый характер.

Используйте фиксированный порядок байтов для обмена данными. И все-таки решение существует. Запишите байты в каноническом порядке, используя переносимый код:

```
unsigned short x;
putchar(x >> 8);        /* запись старшего байта */
putchar(x & 0xFF);      /* запись младшего байта */
```

Затем считайте их обратно по одному и снова соберите в единое целое:

```
unsigned short x;
x = getchar() << 8;     /* чтение старшего байта */
x |= getchar() & 0xFF; /* чтение младшего байта */
```

Этот подход обобщается и на структуры, если записывать их элементы в определенном порядке, по одному байту и без добавления заполнителей. Неважно, какой именно порядок байтов выбран; важно лишь единообразие. Единственное требование заключается в том, чтобы отправитель и получатель согласовали между собой порядок байтов при передаче и количество байтов в каждом объекте. В следующей главе будет продемонстрировано несколько функций для пакетирования абстрактных данных.

Побайтовая обработка может показаться слишком затратной, но учитывая сложность операций ввода-вывода, делающих подобное пакетирование данных необходимым, экономия составляет целые минуты. Взять хотя бы систему X Window, в которой клиент записывает данные в своем естественном порядке, а сервер должен распаковывать все, что ему присылает клиент. На стороне клиента таким образом можно сэкономить несколько байт, однако сервер становится более громоздким и сложным из-за необходимости обрабатывать несколько разных порядков байт (ведь могут существовать как “остроконечные”, так и “тупоконечные” клиенты). Затраты на разработку такого сложного серверного кода намного превзойдут объем экономии со стороны клиентов. Кроме того, это графическая среда, в которой дополнительные затраты на пакетирование байтов — ничто по сравнению с объемом графических операций, которые закодированы этими байтами.

Система X Window запрашивает порядок байтов у клиента и требует от сервера обрабатывать любой из возможных. В противоположность ей, операционная система Plan 9 задает порядок для сообщений, поступающих на файловый (или графический) сервер, а сами данные кодируются в пакеты и декодируются переносимым кодом, как описано выше. На практике разница в быстродействии не заметна; по сравнению с затратами на операции ввода-вывода пакетирование байтов не имеет никакого существенного веса.

Язык Java поддерживает более высокий уровень абстрагирования, чем C или C++, так что в нем вопросы порядка байтов вообще скрыты от программиста. В библиотеках имеется интерфейс `Serializable`, который задает правила для пакетирования элементов данных с целью обмена или пересылки.

Однако в C или C++ всю работу приходится делать самостоятельно. Ключевым преимуществом побайтовой обработки является то, что проблема решается без ди-

ректив #ifdef во всех системах, где байты содержат 8 бит. В следующей главе мы еще остановимся на этом вопросе.

И все же лучшим решением часто является преобразование данных в текстовый формат, который абсолютно переносим (если не считать проблемы CRLF) и не содержит двусмысленности в представлении данных. Но это решение не абсолютно и годится не для всех случаев. Например, может играть роль время выполнения или затраты памяти, а некоторые данные, в частности вещественные с плавающей точкой, могут терять точность из-за округления при прохождении через функции printf и scanf. Если необходимо обмениваться точными вещественнозначными данными, обзаведитесь библиотекой ввода-вывода с хорошим форматированием. Такие библиотеки существуют, хотя и не всегда входят в состав стандартных сред. Особенно тяжело представить числа с плавающей точкой в переносимом двоичном виде, однако при тщательном форматировании их преобразование в текст решает эту проблему.

Имеется одна небольшая проблема переносимости при использовании стандартных функций по работе с двоичными файлами — такие файлы необходимо открывать в двоичном режиме.

```
FILE *fin;
fin = fopen(binary_file, "rb");
c = getc(fin);
```

Если опустить 'b', в системах Unix обычно ничего не случается, а вот в Windows первый же байт Ctrl-Z (восьмеричный код 032, шестнадцатеричный 1A) во входном потоке прекратит чтение, как это случалось на наших глазах с программой работы со строками из главы 5. С другой стороны, использование двоичного режима для считывания текстовых файлов приведет к сохранению символов \r во входных данных, а не к генерированию их на выходе.

8.7. Переносимость и модернизация

Одним из самых неприятных источников проблем с переносимостью является системное программное обеспечение, изменяющееся с течением времени. Изменения могут случиться в любом интерфейсе системы, вызывая совершенно необоснованную несовместимость между существующими версиями программ.

Измените имя, если изменилась спецификация. Наш любимый (если можно так выразиться) пример — это изменение свойств команды echo из системы Unix, первоначальное назначение которой состояло в выводе ее аргументов на экран:

```
% echo hello, world
hello, world
%
```

Однако со временем команда echo стала ключевым элементом многих командных сценариев, и возникла настоятельная потребность генерировать форматированные выходные данные. Поэтому свойства команды echo изменили так, чтобы она *интерпретировала* свои аргументы почти как функция printf:

```
% echo 'hello\nworld'
hello
world
%
```

Эта новая возможность довольно полезна, однако она вызывает проблемы с переносимостью любого сценария на командном языке, который не использует команду echo ни для чего больше, как только для отображения аргументов. Так, поведение следующей команды теперь зависит от используемой версии echo:

```
% echo $PATH
```

Если переменная случайно содержит символ обратной косой черты, что вполне может случиться в среде DOS или Windows, этот символ будет интерпретироваться командой echo. Разница будет похожа на различие между результатами выполнения операторов `printf(str)` и `printf("%s", str)` в том случае, если строка `str` содержит знак процента.

Мы рассказали только малую толику из всей истории о команде echo, но основная проблема обрисовалась достаточно четко: изменения в системе могут привести к созданию различных версий программного обеспечения, которые *намеренно* ведут себя по-разному и при этом *нечаянно* создают проблемы с переносимостью. Обойти эти проблемы довольно трудно. В описанном случае все было бы гораздо проще, если бы новая версия команды echo имела другое имя.

Рассмотрим более непосредственный пример — команду sum системы Unix, выводящую на экран размер и контрольную сумму файла. Ее создали для контроля передачи информации:

```
% sum file
52313 2 file
%
% скопировать file на другую машину othermachine
%
% telnet othermachine
$
$ sum file
52313 2 file
$
```

После пересылки файла контрольная сумма осталась той же, поэтому можно быть уверенными, что старая и новая копии идентичны.

Системы росли и развивались, плодились новые их версии, и вот кто-то заметил, что алгоритм вычисления контрольной суммы далек от совершенства, так что сумму модифицировали и применили лучший алгоритм. Кто-то другой сделал то же наблюдение и еще раз усовершенствовал алгоритм. В итоге сегодня мы имеем несколько разных версий утилиты sum, и все они дают разные ответы. Мы скопировали один файл на ближайшие компьютеры, чтобы посмотреть результат работы sum:

```
% sum file
52313 2 file
%
% копировать file в систему machine2
% копировать file в систему machine3
```

```
% telnet machine2
$
$ sum file
eaa0d468    713     file
$ telnet machine3
>
> sum file
62992   1   file
>
```

Итак, чем же вызвана разница между контрольными суммами: повреждением файла или различием версий sum? Возможно, и тем и другим.

Получается, что команда sum является просто эталоном непереносимости: программа, предназначавшаяся для контроля копирования файлов с одного компьютера на другой, стала совершенно бесполезной для своей исходной цели благодаря наличию множества разных версий.

Исходная утилита sum вполне подходила для своей простой задачи, и ее несложный алгоритм вычисления контрольной суммы оправдывал себя. "Усовершенствование", возможно, и улучшило программу в некоторой степени, но не настолько, чтобы оправдать этим потерю совместимости. Проблема заключается не в самих усовершенствованиях, а в том, что несколько несовместимых программ фигурирует под одним именем. Внесенные изменения породили проблему с версиями, которая будет докучать пользователям еще многие годы.

Поддерживайте совместимость с существующими программами и данными. Когда в свет выходит новая версия пользовательской программы наподобие текстового редактора, обычно она способна читать файлы, созданные предыдущими, более старыми версиями. Это вполне естественно: в программу добавляются новые, ранее не ожидавшиеся возможности, и формат данных должен эволюционировать. Однако в новых версиях иногда забывают обеспечить пользователей средствами для записи файлов в предыдущих форматах. Пользователи новой версии, даже если они не пользуются ее новыми возможностями, не могут обмениваться файлами с теми, у кого есть только старая версия, поэтому все они под давлением обстоятельств вынуждены переходить на новую версию. Это неприятно и достойно всеобщего осуждения, будь то технический просчет или специальная маркетинговая стратегия.

Обратной совместимостью называется соответствие программы ее старой спецификации. Если вы собираетесь вносить в программу изменения, постарайтесь сохранить полную поддержку старой версии и соответствующих данных. Хорошенько задокументируйте изменения и обеспечьте путь к отступлению, т.е. к восстановлению исходных функций. И что наиболее важно, взвесьте, является ли предложенное изменение действительно усовершенствованием, перевешивающим недостаток плохой переносимости, если таковая вносится данным изменением в программу.

8.8. Интернационализация

Живя в Соединенных Штатах, легко забыть, что английский — не единственный язык в мире, ASCII — не единственный набор символов, $ — не единственный символ денежной единицы, даты могут записываться начиная с дня, время может обозначаться по 24-часовой шкале и т.д. Таким образом, еще один аспект широко понимаемой переносимости заключается в возможности переноса программ через языковые и культурные барьеры. Эта тема практически необъятна, но здесь мы наметим только некоторые основные ее вопросы.

Если программу необходимо приспособить для выполнения в любой среде без предположений о ее национально-культурном характере, то говорят об *интернационализации* программы. С ней связано много проблем, от символьных наборов до интерпретации пиктограмм в графических интерфейсах.

Не предполагайте, что используется кодировка ASCII. В большинстве стран мира используются более обширные, чем ASCII, символьные наборы. Стандартные функции анализа символов, объявленные в файле `ctype.h`, обычно скрывают эти различия от программиста. Следующая проверка не зависит от конкретной кодировки:

```
if (isalpha(c)) ...
```

Кроме того, она работает и в тех языковых средах, где алфавитных символов может быть меньше или больше, чем в латинском наборе от `a` до `z`, если программа скомпилирована в этой среде. Конечно, само название `isalpha` ("является ли алфавитным") достаточно красноречиво — в некоторых языках нет алфавитов, и о такой проверке вообще не идет речь.

В большинстве европейских стран кодировка ASCII расширена, поскольку она строго определяет коды только до `0x7F` (т.е. с использованием первых 7 бит), оставляя дополнительные символы для представления букв других языков. Кодировка Latin-1, повсеместно используемая в Западной Европе, является надмножеством ASCII, в котором байтовые коды от `80` до `FF` используются для разных символических знаков и букв с надстрочными или подстрочными знаками; например, `E7` представляет букву ç. Английское слово `boy` ("мальчик, парень") представляется в кодировке ASCII или Latin-1 тремя байтами с шестнадцатеричными значениями `62 6F 79`, тогда как французское слово `garçon` (с тем же значением) представляется в кодировке Latin-1 байтами `67 61 72 E7 6F 6E`. В других языках определяются другие знаки и символы, но все они не могут поместиться в 128 кодов, не используемых кодировкой ASCII. Поэтому существует целый ряд конфликтующих стандартов, определяющих символы с кодами от `80` до `FF`.

Некоторые языки вообще не помещаются в 8-битовое пространство кодировки. В основных восточноазиатских языках используются тысячи символов. Так, кодировки, используемые в Китае, Японии и Корее, основаны на 16-битовых представлениях. В результате чтение документа, написанного на одном языке, на компьютере, локализованном под другой язык, представляет собой серьезную проблему. Даже предполагая, что символы не искажаются при переносе документа, чтение китайскоязычного документа на американском компьютере требует как минимум установки специального программного обеспечения и шрифтов. Если же приходится

совместно пользоваться китайским, английским и русским языками, то возникающие сложности просто неописуемы.

Символьный набор Unicode представляет собой попытку исправить ситуацию, создав единую кодировку для всех языков мира. Кодировка Unicode, совместимая с 16-разрядным подмножеством стандарта ISO 10646, основана на использовании 16 битов для каждого символа, причем коды до 00FF включительно соответствуют кодировке Latin-1. Таким образом, слово `garçon` в этой кодировке представляется 16-битовыми кодами `0067 0061 0072 00E7 006F 006E`, кириллические символы занимают диапазон кодов от `0401` до `04FF`, а иероглифической письменности отведен большой блок, начинающийся с `3000`. Все распространенные языки, как и множество менее распространенных, представлены в кодировке Unicode, так что именно ею следует пользоваться при пересылке документов между разными странами или для хранения документов на нескольких языках. Кодировка Unicode становится все более популярной в Internet, а некоторые системы даже поддерживают ее как стандартный формат. Например, в языке Java именно Unicode является естественным символьным форматом, используемым в текстовых строках. Операционные системы Plan 9 и Inferno целиком основаны на Unicode; это относится даже к именам файлов и пользователей. Система Microsoft Windows поддерживает символьный набор Unicode, но он не является в ней обязательным. Большинство приложений для Windows по-прежнему работает лучше всего с кодировкой ASCII, однако на практике происходит быстрая эволюция в сторону Unicode.

Все же с кодировкой Unicode связана и проблема: символы больше не помещаются в один байт, так что текст подвержен путанице с порядком байтов. Чтобы избежать этого, перед пересылкой между программами или по сети документы Unicode обычно транслируются в потоково-байтовую кодировку под названием UTF-8. Для передачи каждый 16-разрядный символ кодируется последовательностью из 1, 2 или 3 байт. Символьный набор ASCII занимает диапазон кодов от 00 до 7F, причем все эти коды помещаются в один байт и именно так представляются в UTF-8, что обеспечивает обратную совместимость этой кодировки с ASCII. Значения от 80 до 7FF представляются двумя байтами, а 800 и выше — тремя байтами. Таким образом, слово `garçon` выглядит в кодировке UTF-8 как последовательность байтов `67 61 72 C3 A7 6F 6E`; код Unicode `E7`, т.е. символ ç, представляется в UTF-8 двумя байтами `C3 A7`.

Обратная совместимость кодировок UTF-8 и ASCII — великое благо, поскольку она позволяет программам, воспринимающим текст как неинтерпретируемый поток байтов, работать с Unicode-текстом на любом языке. Мы испытали программу `markov` из главы 3 на тексте в кодировке UTF-8 на русском, греческом, японском и китайском языках, и всякий раз она выполнялась без проблем. Для европейских языков, в которых слова отделяются пробелами, символами табуляции или символами конца строки с кодами ASCII, на выходе получалась забавная словесная чепуха, как и ожидалось. Для других языков неплохо было бы изменить правила разбиения строк на слова, чтобы получить выходные данные, более соответствующие исходному предназначению программы.

Языки C и C++ поддерживают “расширенные символы”, представляющие собой 16-разрядные или более длинные целые числа, а также несколько вспомогательных функций для обработки символов Unicode или другого большого символьного набора. Литералы, состоящие из расширенных символов, записываются так: `L"..."`. Увы, с ними связаны дополнительные проблемы переносимости: программу с литеральными константами из расширенных символов можно прочитать только на дисплее, поддерживающем этот набор.

Поскольку для согласованной передачи из одной системы в другую символы приходится конвертировать в байтовые потоки (например, UTF-8), в языке C есть функции для преобразования расширенных символов в байт-коды и наоборот. Но каково именно это преобразование? Правила интерпретации символьного набора и определение их перекодировки в поток байтов спрятаны в библиотеках, и извлечь их оттуда не так-то просто — ситуация, мягко говоря, неудовлетворительная. Хотелось бы надеяться, что когда-нибудь в светлом будущем все окончательно примут к использованию единый символьный набор; однако более вероятный сценарий развития событий состоит в том, что путаница, связанная с порядком байтов, еще долго будет терроризировать мир программирования.

Не предполагайте использование английского языка. Разработчики интерфейсов должны иметь в виду, что для выражения одной и той же мысли в разных языках используется самое разное количество символов, поэтому в массивах и на экране должно быть предусмотрено достаточно пространства для этого.

Возьмем, например, сообщения об ошибках. Как минимум, в них не должен употребляться сленг и какие бы то ни было выражения из профессионального арго, понятные только избранным. Такие сообщения следует с самого начала формулировать простым, понятным языком. Распространенный подход — это собрать текст всех сообщений в одном месте, чтобы при переводе на другой язык их можно было легко заменить.

Имеется множество других культурных различий, например формат мм/дд/гг для дат, используемый только в Северной Америке. Если есть вероятность, что программа будет использоваться в другой стране, такую зависимость от культурных традиций следует минимизировать или вообще исключить. Пиктограммы в графических интерфейсах также часто зависят от культурного контекста. Некоторые из них могут оказаться совершенно непонятными пользователям той среды, для которой предназначается программа, не говоря уже об остальных.

8.9. Резюме

Легко переносимый код — это идеал, за который стоит бороться, поскольку в противном случае очень много времени тратится на внесение изменений при переносе программы из одной системы в другую или на поддержание работоспособности программы при эволюции ее системного окружения или ее самой. Однако переносимость не достается бесплатно. Она требует тщательного программирования и знания проблемных мест совместимости во всех системах, в которые планируется перенос программы.

Были рассмотрены два подхода к переносимости: “объединение” и “пересечение”. “Объединение” подразумевает разработку таких программ, которые работают в каждой из намеченных систем благодаря тому, что в них включен весь необходимый для

этого код с использованием таких механизмов, как условная компиляция. Этот подход имеет множество недостатков: слишком большие объемы кода, нередко слишком высокая его сложность, трудность доработки и тестирования программ.

"Пересечение" подразумевает написание как можно большей части кода таким образом, чтобы она работала без изменений в любой из систем. Системно-зависимые фрагменты, без которых не обойтись, инкапсулируются в отдельные файлы исходного кода, служащие интерфейсом между программой и конкретной операционной средой. Этот подход также имеет свои недостатки, среди которых — потенциальная потеря быстродействия и даже отдельных функциональных возможностей, но в дальней перспективе его преимущества окупают затраты.

Дополнительная литература

Существует много описаний языков программирования, но лишь немногие из них достаточно точны и полны, чтобы служить надежными справочниками. Авторы питают субъективное пристрастие к книге Brian Kernighan, Dennis Ritchie, *The C Programming Language* (Prentice Hall, 1988), но и она не может служить заменой стандарту. Книга Sam Harbison, Guy Steele, *C: A Reference Manual* (Prentice Hall, 1994), к этому моменту вышедшая в четвертом издании, содержит хорошие рекомендации по переносимости программ на C. Официальные стандарты языков C и C++ можно получить от ISO (*International Organization for Standardization* — Международная организация по стандартизации). Ближайшим к стандарту описанием Java можно считать книгу James Gosling, Bill Joy, Guy Steele, *The Java Language Specification* (Addison-Wesley, 1996).

Книга Rich Stevens, *Advanced Programming in the Unix Environment* (Addison-Wesley, 1992) рекомендуется для изучения всеми программистами, использующими Unix. Она содержит обстоятельное изложение принципов переноса программ между различными вариантами Unix.

POSIX (*Portable Operating System Interface* — переносимый интерфейс операционных систем) представляет собой международный стандарт, определяющий команды и библиотеки на базе Unix. Он обеспечивает единую стандартную среду, переносимость исходного кода приложений, а также единый интерфейс ввода-вывода, файловых систем и процессов. Этот стандарт описан в серии книг, опубликованных организацией IEEE.

Термины "тупоконечники" и "остроконечники" ввел Джонатан Свифт в 1726 г. в своих "Путешествиях Гулливера". Статья Danny Cohen, "On holy wars and a plea for peace", *IEEE Computer*, October 1981, представляет собой волшебную сказку о проблемах порядка байтов. Именно в ней упомянутые термины впервые появляются в контексте вычислительной техники.

Система Plan 9, разработанная в лабораториях Bell, сделала переносимость центральным приоритетом разработчиков. Система компилируется из одного исходного кода, лишенного директив `#ifdef`, на целом ряде процессоров. В ней повсеместно используется символьный набор Unicode. Последние версии программы Sam (впервые описанной в статье "The Text Editor `sam`", *Software—Practice and Experience*, 1987, **17**, 11, p. 813–845) также используют набор Unicode и при этом работают в самых разных системах. Проблемы работы с 16-разрядными наборами наподобие Unicode рассматриваются в статье Rob Pike, Ken Thompson, "Hello World...", *Proceedings of the Winter*

1993 USENIX Conference, San Diego, 1993, p. 43–50. Кодировка UTF-8 впервые появилась на свет именно в этой статье. Данная работа имеется на Web-сайте проекта Plan 9 при Bell Labs вместе с текущей версией программы Sam.

Система Inferno, основанная на опыте проекта Plan 9, в некоторой степени аналогична Java по принципам устройства. Она определяет виртуальную машину, которую можно реализовать в любой реальной системе, предоставляет язык (Limbo), транслируемый в инструкции виртуальной машины, и использует Unicode в качестве своего базового символьного набора. Она также включает в себя виртуальную операционную систему, обеспечивающую переносимый интерфейс для целого ряда коммерческих систем. Ее описание можно найти в статье Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard W. Trickey, Philip Winterbottom, “The Inferno Operating System”, *Bell Labs Technical Journal*, 1997, **2**, 1.

9

Системы обозначений

Вероятно, из всех творений человека самое поразительное — это язык.

Джайлз Литтон Стречи. "Словесность и поэзия"
(Giles Litton Strachey, Words and Poetry*)*

Именно правильный выбор языка главным образом определяет, насколько легко будет писать программу. Вот почему в арсенале программиста-практика должны быть не только языки общего назначения наподобие C и его производных, но также программные среды-оболочки, языки разработки сценариев и множество проблемно-ориентированных языков.

Сила правильно выбранной системы обозначений простирается далеко за пределы традиционного программирования в область специализированных задач. Так, с помощью регулярных выражений можно записывать компактные (хотя нередко и трудночитаемые) определения классов строк; язык HTML позволяет определять макеты интерактивных документов и внедрять в них программы на других языках, например JavaScript; язык PostScript дает возможность выразить целый документ, такой как эта книга, в виде программы типографского макета. Программы текстового редактирования и электронных таблиц часто содержат средства программирования на таких языках, как Visual Basic, для вычисления выражений, получения информации или управления макетом документа.

Если для выполнения самой рутинной работы приходится писать слишком много кода или не удается выразить ту или иную процедуру достаточно удобно, возможно, вы просто пользуетесь не тем языком. Если нужный язык еще не существует, вам выпадает прекрасная возможность создать его самостоятельно. Изобретение языка совсем не означает разработку нового Java; часто решение неудобной задачи становится вполне прозрачным после смены системы обозначений. Для примера возьмем строки формата в функциях семейства `printf`, представляющие собой компактный и выразительный способ управления выводом на экран различных значений.

В этой главе мы поговорим о том, как изменение системы записи или обозначений позволяет решать проблемы, и продемонстрируем, как можно реализовать свой собственный специализированный язык. Мы даже рассмотрим возможность разработки такой программы, которая бы сама писала другую программу. Это доведенный до крайности случай использования альтернативной системы обозначений, который встречается чаще и при этом гораздо проще в реализации, чем думают многие программисты.

9.1. Форматирование данных

Всегда существует разрыв между тем, что мы хотим приказать компьютеру ("реши-ка мою проблему"), и тем, что от нас требуется приказать, чтобы работа была выполнена. Чем меньше этот разрыв, тем лучше. Удачно выбранная система обозначений позволяет проще выразить то, чего мы хотим, и мешает сказать что-нибудь не то по ошибке. Иногда хорошие обозначения открывают новые перспективы и позволяют решать ранее неразрешимые задачи, или даже ведут к новым горизонтам знания.

Существуют системы обозначений для узких областей специализации. Будем называть их *мини-языками*. Они не только обеспечивают удобный интерфейс, но и помогают в организации программ, которые их реализуют. Хорошим примером мини-языка являются управляющие строки функции `printf`:

```
printf("%d %6.2f %-10.10s\n", i, f, s);
```

Каждый знак `%` в строке формата сигнализирует о том, что необходимо считать, проинтерпретировать и вывести очередной аргумент функции; после нескольких необязательных флагов и показателей ширины полей тип параметра указывается буквой. Эта запись компактна, интуитивно естественна и проста в составлении, а ее реализация не представляет особых трудностей. Альтернативы, принятые в C++ (`iostream`) и Java (`java.io`), несколько более неуклюжи, поскольку в них нет специальной системы обозначений. Правда, они расширяются до пользовательских типов данных и предлагают проверку соответствия типов.

Некоторые нестандартные реализации `printf` позволяют добавлять новые преобразования к встроенному набору. Это удобно, если у вас имеются другие типы данных, требующие форматирования при выводе. Например, компилятор мог бы использовать спецификацию `%L` для номера строки и имени файла; в графической системе может применяться спецификация `%P` для точки и `%R` для прямоугольника. В том же духе составлена и неудобочитаемая строка из букв и цифр для получения биржевых котировок, рассмотренная в главе 4. Это компактная запись для организации биржевых данных.

Подобные примеры можно составить также на C и C++. Предположим, нужно переслать пакеты смешанных данных различных типов из одной системы в другую. Как было сказано в главе 8, самое надежное решение — это преобразовать данные в текстовое представление. Однако для стандартного сетевого протокола формат скорее должен быть двоичным из соображений быстродействия или объема. Как же написать код для работы с пакетами так, чтобы он был переносимым, высокоэффективным и простым в использовании?

Чтобы конкретизировать наше изложение, представим себе задачу пересылки пакетов из 8-, 16- и 32-битовых элементов данных между системами. ANSI C требует, чтобы в переменной типа `char` всегда хранилось не менее 8 бит, в `short` — не менее 16, а в `long` — не менее 32. Именно этими типами мы будем пользоваться для представления наших данных. Типов пакетов будет много; так, тип 1 может содержать однобайтовый спецификатор типа, двухбайтовый счетчик, однобайтовое кодовое значение и четырехбайтовый элемент данных:

0x01	cnt_1	cnt_0	val	$data_3$	$data_2$	$data_1$	$data_0$

Пакет типа 2 может содержать одно короткое слово данных (`short`) и два длинных (`long`):

0x02	cnt_1	cnt_0	$dw1_3$	$dw1_2$	$dw1_1$	$dw1_0$	$dw2_3$	$dw2_2$	$dw2_1$	$dw2_0$

Один из подходов — это написать функции кодирования и декодирования для всех возможных типов пакетов:

```
int pack_type1(unsigned char *buf, unsigned short count,
        unsigned char val, unsigned long data)
{
    unsigned char *bp;

    bp = buf;
    *bp++ = 0x01;
    *bp++ = count >> 8;
    *bp++ = count;
    *bp++ = val;
    *bp++ = data >> 24;
    *bp++ = data >> 16;
    *bp++ = data >> 8;
    *bp++ = data;
    return bp - buf;
}
```

В реальном протоколе таких функций, представляющих всевозможные "вариации на тему", были бы десятки. Эти процедуры можно упростить, используя макроопределения или функции для обработки элементарных типов данных (`short`, `long` и т.д.), но даже в этом случае такой повторяющийся код подвержен ошибкам, трудно читается и плохо поддается доработке.

Присущая коду повторяемость наводит на мысль, что нам может помочь удачно выбранная система обозначений. Заимствуя идею у `printf`, можно определить мини-язык, на котором каждый пакет будет описываться короткой строкой, полностью определяющей его структуру. Последовательные элементы пакета могут кодироваться обозначением `c` для 8-битового символа, `s` для 16-битового короткого целого (`short`) и `l` для 32-битового длинного целого (`long`). Например, представленный выше тип пакета 1, включая его начальный байт типа, можно описать строкой формата `cscl`. В этом случае можно написать одну функцию `pack` для создания пакетов любого типа. Данный пакет создавался бы следующим вызовом:

```
pack(buf, "cscl", 0x01, count, val, data);
```

Поскольку наша строка формата содержит только спецификации данных, нет никакой необходимости в знаке `%`, используемом в `printf`.

На практике информация, приведенная в начале пакета, обычно сообщает получателю, как декодировать остальные данные. Мы предполагаем, что первый байт пакета определяет его организацию. Отправитель кодирует данные в этот формат и посылает их; получатель считывает пакет, извлекает первый байт и пользуется им для декодирования остальной информации.

Ниже приведена реализация функции `pack`, заполняющей буфер `buf` закодированным представлением ее аргументов, которое определяется заданным форматом. Мы сделали все значения беззнаковыми, в том числе и байты в буфере пакета, чтобы избежать проблем с расширением знака. Мы также воспользовались традиционными операторами `typedef` для краткости объявлений:

```
typedef unsigned char  uchar;
typedef unsigned short ushort;
typedef unsigned long  ulong;
```

Подобно `sprintf`, `strcpy` и другим схожим функциям, в `pack` подразумевается, что буфер достаточно велик для хранения результата; ответственность за это целиком лежит на вызывающем модуле. Не предпринимается также никаких попыток проверить соответствие между строкой формата и списком аргументов.

```
#include <stdarg.h>

/* pack: пакетирует двоичные данные в buf, возвращает длину */
int pack(uchar *buf, char *fmt, ...)
{
    va_list args;
    char *p;
    uchar *bp;
    ushort s;
    ulong l;
    bp = buf;
    va_start(args, fmt);
    for (p = fmt; *p != '\0'; p++) {
        switch (*p) {
        case 'c': /* char */
            *bp++ = va_arg(args, int);
            break;
        case 's': /* short */
            s = va_arg(args, int);
            *bp++ = s >> 8;
            *bp++ = s;
            break;
        case 'l': /* long */
            l = va_arg(args, ulong);
            *bp++ = l >> 24;
            *bp++ = l >> 16;
            *bp++ = l >> 8;
            *bp++ = l;
            break;
        default: /* неопознанный символ типа */
            va_end(args);
            return -1;
        }
    }
    va_end(args);
    return bp - buf;
}
```

Функция pack использует заголовочный файл stdarg.h более активно, чем это делала функция eprintf (см. главу 4). Последовательные аргументы извлекаются с использованием макроса va_arg, причем первый операнд является переменной типа va_list, инициализированной путем вызова va_start, а второй операнд — типом аргумента (вот почему va_arg является макросом, а не функцией). После окончания обработки следует вызвать va_end. Хотя спецификации 'c' и 's' представляют аргументы типа char и short соответственно, их необходимо извлекать в формате int, поскольку компилятор C автоматически расширяет аргументы типа char и short до int, если они представлены аргументом-многоточием.

Каждая функция pack_type теперь становится тривиальной и содержит всего одну строку, формируя вызов pack на основе своих аргументов:

```
/* pack_type1: создает пакет формата 1 */
int pack_type1(uchar *buf, ushort count, uchar val, ulong data)
{
    return pack(buf, "cscl", 0x01, count, val, data);
}
```

Для декодирования пакета можно сделать то же самое: вместо того чтобы писать отдельный код для каждого формата, можно вызывать одну и ту же функцию unpack с разными строками формата. Это позволяет сосредоточить преобразование в одном месте:

```
/* unpack: декодирует элементы пакета buf, возвращает длину */
int unpack(uchar *buf, char *fmt, ...)
{
    va_list args;
    char *p;
    uchar *bp, *pc;
    ushort *ps;
    ulong *pl;
    bp = buf;
    va_start(args, fmt);
    for (p = fmt; *p != '\0'; p++) {
        switch (*p) {
        case 'c': /* char */
            pc = va_arg(args, uchar*);
            *pc = *bp++;
            break;
        case 's': /* short */
            ps = va_arg(args, ushort*);
            *ps = *bp++ << 8;
            *ps |= *bp++;
            break;
        case 'l': /* long */
            pl = va_arg(args, ulong*);
            *pl = *bp++ << 24;
            *pl |= *bp++ << 16;
            *pl |= *bp++ << 8;
            *pl |= *bp++;
            break;
        default:  /* неопознанный символ типа */
            va_end(args);
            return -1;
```

```
        }
    }
    va_end(args);
    return bp - buf;
}
```

Как и scanf, функция unpack должна возвращать несколько значений в вызывающий модуль, поэтому ее аргументы — это указатели на переменные, в которые следует поместить результаты. Возвращаемое значение равно количеству байт в пакете; его можно использовать для контроля.

Поскольку все значения не имеют знака и мы обходимся только стандартными размерами ANSI C для типов данных, передача информации происходит в полностью совместимом режиме даже между системами с разными размерами типов short и long. Информация будет передана корректно при условии, что программа, использующая pack, не попытается пересылать в виде long (к примеру) значение, не представляемое 32 битами. Фактически пересылаются младшие 32 бита этого значения. Если необходимо пересылать более длинные данные, придется определить еще один формат.

Теперь уже легко написать функции декодирования данных из пакетов, использующие функцию unpack:

```
/* unpack_type2: декодирует пакет типа 2 */
int unpack_type2(int n, uchar *buf)
{
    uchar c;
    ushort count;
    ulong dw1, dw2;
    if (unpack(buf, "csll", &c, &count, &dw1, &dw2) != n)
        return -1;
    assert(c == 0x02);
    return process_type2(count, dw1, dw2);
}
```

Для вызова функции unpack_type2 вначале следует распознать, что перед нами пакет типа 2. Это подразумевает цикл приема пакетов наподобие следующего:

```
while ((n = readpacket(network, buf, BUFSIZ)) > 0) {
    switch (buf[0]) {
    default:
        eprintf("bad packet type 0x%x", buf[0]);
        break;
    case 1:
        unpack_type1(n, buf);
        break;
    case 2:
        unpack_type2(n, buf);
        break;
    ...
    }
}
```

Подобный код может сильно разрастись. Более компактный метод состоит в том, чтобы определить таблицу указателей на функции декодирования с индексацией по типу:

```
int (*unpackfn[])(int, uchar *) = {
    unpack_type0,
    unpack_type1,
    unpack_type2,
};
```

Каждая функция из таблицы анализирует пакет, проверяет результат и инициирует дальнейшие операции по обработке этого пакета. Наличие таблицы делает работу приемного модуля тривиальной:

```
/* receive: считывает пакеты из сети, задает обработку */
void receive(int network)
{
    uchar type, buf[BUFSIZ];
    int n;
    while ((n = readpacket(network, buf, BUFSIZ)) > 0) {
        type = buf[0];
        if (type >= NELEMS(unpackfn))
            eprintf("bad packet type 0x%x", type);
        if ((*unpackfn[type])(n, buf) < 0)
            eprintf("protocol error, type %x length %d",
                type, n);
    }
}
```

Код для обработки отдельных пакетов компактен, сосредоточен в одном месте и легко поддается доработке. Приемный модуль достаточно независим от самого протокола, устроен просто и работает быстро.

Этот пример основан на реальном коде, предназначенном для коммерческого сетевого протокола. Как только автор осознал работоспособность этого подхода, сразу же несколько тысяч повторяющихся строк, сильно подверженных ошибкам, сократились до нескольких сотен строк аккуратного, легко дорабатываемого кода. Итак, выбор удачной системы обозначений позволил значительно упростить работу.

Упражнение 9.1. Модифицируйте функции `pack` и `unpack` для корректной передачи значений со знаком, даже между системами с разными длинами типов `short` и `long`. Как следует модифицировать строки формата, чтобы в них указывались типы со знаком? Как можно протестировать это код, например, на правильность передачи числа `-1` с компьютера, на котором числа типа `long` имеют длину 32 бита, на компьютер с 64-разрядными числами этого типа?

Упражнение 9.2. Доработайте функции `pack` и `unpack` для работы со строками; один из вариантов — это включить длину строки данных в строку формата. Расширьте возможности этих функций так, чтобы они могли работать с повторяющимися элементами посредством счетчика. Как это связано с кодированием строк?

Упражнение 9.3. Таблица указателей на функции в программе на C очень близка по своей идее к механизму виртуальных функций C++. Перепишите `pack`, `unpack` и `receive` на C++ так, чтобы воспользоваться этим удобным средством.

Упражнение 9.4. Напишите аналог функции printf в виде программы с интерфейсом командной строки, которая выводит второй и последующие аргументы в формате, задаваемом первым аргументом. В некоторых средах программирования такая возможность уже существует как встроенная.

Упражнение 9.5. Напишите функцию для реализации спецификаций формата, имеющихся в программах электронных таблиц или классе DecimalFormat языка Java. Эти спецификации служат для отображения чисел по шаблонам, указывающим обязательные и необязательные цифры, местоположение десятичной точки и запятых, и т.д. Ниже продемонстрирован этот формат:

```
##,##0.00
```

Приведенный формат задает число с двумя десятичными позициями, как минимум одной цифрой слева от десятичной точки, запятой после числа тысяч и заполнителем вплоть до количества десятков тысяч. В этом формате число 12345.67 представляется как 12,345.67, а .4 — в виде ______0.40 (если подставить знаки подчеркивания вместо пробелов). Чтобы получить точную спецификацию, изучите определение класса DecimalFormat или описание программы работы с электронными таблицами.

9.2. Регулярные выражения

Спецификации формата для функций pack и unpack образуют очень простую систему обозначений для определения состава сетевых пакетов. Наша следующая тема — несколько более сложная, но и намного более выразительная система обозначений под названием *регулярные выражения*, служащие для задания текстовых образцов и шаблонов. Регулярные выражения уже использовались в нашей книге время от времени без строгого определения; однако они достаточно знакомы многим и понятны без разъяснений. Хотя регулярные выражения очень распространены в среде программирования Unix, в других системах они используются не так часто. В этом разделе будут продемонстрированы некоторые из их богатых возможностей. На тот случай, если у вас под рукой нет библиотеки для работы с регулярными выражениями, будет показана также ее рудиментарная реализация.

Существует несколько разновидностей регулярных выражений, однако по сути они все одинаковы. Это способы описания образцов текста, состоящие из литеральных символов, а также условных знаков для повторений, альтернатив и определенных классов символов наподобие букв или цифр. Один из известных примеров — так называемые символы подстановки, используемые в командных строках или оболочках для обозначения групп имен файлов. Обычно звездочка (*) означает "любая строка символов". Возьмем следующую команду:

```
C:\> del *.exe
```

В ней используется шаблон, который соответствует всем файлам с произвольными именами и расширением .exe. Как это часто бывает, подробности реализации варьируются от системы к системе и даже от программы к программе.

Хотя причуды различных программ и систем заставляют думать, что регулярные выражения — это нестандартное проблемно-ориентированное средство, на самом деле это язык с формализованной грамматикой и точной семантикой для каждого высказывания данного языка. Более того, при хорошей реализации они могут работать очень быстро, поскольку сочетание строгой теории и инженерной практики обычно дает отличные результаты. Это пример преимуществ специализированных алгоритмов, на которые мы намекали в главе 2.

Регулярное выражение — это последовательность символов, определяющая набор строк, который должен ему удовлетворять. Большинство символов выражения являются просто символами и ничем больше, так что регулярное выражение `abc` обозначает только эту строку, где бы она не встречалась. Кроме того, существует ряд ***мета-символов***, обозначающих повторение, группировку или позиционирование. В стандартных регулярных выражениях Unix символ `^` обозначает начало строки, а `$` — ее конец, так что `^x` соответствует символу `x` только в начале строки, `x$` — в конце строки, `^x$` — в строке из единственного символа `x`, а `^$` — в пустой строке.

Символ точки обозначает произвольный символ, так что шаблон `x.y` соответствует строкам `xay`, `x2y` и т.д., но не `xy` или `xaby`, а `^.$` соответствует строке с единственным произвольным символом.

Набор символов в квадратных скобках соответствует любому одному из них, т.е. запись `[0123456789]` обозначает одну произвольную цифру; эту строку можно сократить до `[0-9]`.

Эти элементарные “кирпичики” можно комбинировать с помощью круглых скобок для группировки, знака `|` для выбора между альтернативами, `*` для выбора ни одного или любого количества вхождений, `+` для поиска одного или нескольких вхождений, а также `?` для ни одного или одного вхождений. Наконец, обратная косая черта используется как префикс перед мета-символами, чтобы передать их буквально, без специального значения; поэтому `\*` является литералом, а `\\` — литеральной обратной косой чертой.

Наиболее известная программа для работы с регулярными выражениями — это утилита `grep`, которая уже неоднократно упоминалась ранее. Это образцовый пример ценности хорошей системы обозначений. Программа применяет заданное регулярное выражение к каждой строке своих входных файлов и выводит те строки, которые соответствуют выражению. Эта простая операция в сочетании с мощью регулярных выражений позволяет решать множество повседневных задач. В следующих примерах просим заметить, что синтаксис регулярных выражений, используемых в качестве аргумента `grep`, отличается от символов подстановки, задающих наборы файлов. Это различие отражает разные способы использования. Итак:

Какие из файлов исходного кода используют класс `Regexp`?

```
% grep Regexp *.java
```

Какие файлы содержат его реализацию?

```
% grep 'class.*Regexp' *.java
```

Когда было сохранено почтовое сообщение от Боба?

```
% grep '^From:.* bob@' mail/*
```

Сколько непустых строк исходного кода содержит программа?

```
% grep '.' *.c++ | wc
```

Имея ключи для вывода номеров найденных строк, подсчета совпадений, игнорирования регистра символов, обращения цели поиска (т.е. поиска строк, не удовлетворяющих заданному выражению) и других вариантов реализации основной идеи, программа grep используется так широко, что давно стала классическим инструментом программирования пользовательских командных сценариев.

К сожалению, программа grep или ее эквивалент имеется не в каждой операционной среде. В некоторых системах есть библиотеки для работы с регулярными выражениями, обычно regex или regexp, которыми можно пользоваться при написании программы, аналогичной grep. Если же нет ни того ни другого, можно самостоятельно реализовать скромное подмножество полного языка регулярных выражений. Здесь мы представим реализацию регулярных выражений, а затем и аналога программы grep. Для простоты учтем только мета-символы ^ $., а также *, который обозначает повторение одной предыдущей точки или литерального символа. Это подмножество поддерживает весьма значительную часть возможностей регулярных выражений, тогда как программировать придется лишь очень малую часть от общего объема работы, необходимой для полной их реализации.

Начнем с самой функции определения совпадений. Ее назначение — определить, соответствует ли текстовая строка заданному регулярному выражению.

```
/* match: ищет регулярное выражение в тексте */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do { /* искать даже в пустой строке */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}
```

Если регулярное выражение начинается с символа ^, текст должен давать совпадение с остатком выражения. В противном случае идем по тексту, пользуясь функцией matchhere для поиска совпадений в произвольной позиции. Как только совпадение обнаружено, работа окончена. Отметим использование цикла do-while: выражения могут давать совпадения с пустой строкой (например, $ дает совпадение с пустой строкой в ее конце, а .* подходит под любое количество символов, включая нуль), поэтому функцию matchhere необходимо вызывать, даже если текст пуст.

Большую часть работы выполняет рекурсивная функция matchhere:

```
/* matchhere: ищет выражение regexp в начале текста text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
    if (regexp[0] == '$' && regexp[1] == '\0')
```

```
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}
```

Если регулярное выражение пусто, значит, достигнут конец и найдено совпадение. Если выражение заканчивается на $, оно дает совпадение только в том случае, если текст тоже находится в конце. Если выражение начинается с точки, то оно дает совпадение с любым символом. В противном случае выражение начинается с простого символа, дающего совпадение в тексте только с самим собой. Поэтому символ ^ или $ в середине регулярного выражение должен восприниматься как литерал, а не *мета-символ*.

Обратите внимание, что функция `matchhere` вызывает сама себя после проверки совпадения одного символа из шаблона и строки, поэтому глубина рекурсии может достигать по величине длины шаблона.

Один хитрый случай имеет место тогда, когда выражение начинается с символа со звездочкой, например `x*`. Тогда вызывается `matchstar` с первым аргументом в виде операнда звездочки (`x`) и последующими в виде образца, следующего за звездочкой и текстом:

```
/* matchstar: ищет c*regexp в начале текста */
int matchstar(int c, char *regexp, char *text)
{
    do { /* символ * соответствует любому количеству вхождений */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

Здесь снова используется цикл `do-while`, запускаемый условием, что регулярное выражение `x*` может давать совпадения с нулевым количеством символов. В цикле проверяется, соответствует ли текст оставшейся части выражения. Для этого анализируется каждая позиция в тексте, пока первый символ соответствует операнду звездочки.

Это весьма незамысловатый алгоритм, но он работает, и его объем из менее чем 30 строк кода показывает, что для реализации регулярных выражений нет нужды прибегать к сверхсложной технике программирования.

Вскоре мы представим некоторые новые идеи для расширения возможностей кода. Пока же напишем версию программы `grep`, использующую функцию `match`. Вот главный ее модуль:

```
/* grep main: ищет регулярное выражение в файле */
int main(int argc, char *argv[])
{
    int i, nmatch;
    FILE *f;
    setprogname("grep");
    if (argc < 2)
        eprintf("usage: grep regexp [file ...]");
    nmatch = 0;
```

```
    if (argc == 2) {
        if (grep(argv[1], stdin, NULL))
            nmatch++;
    } else {
        for (i = 2; i < argc; i++) {
            f = fopen(argv[i], "r");
            if (f == NULL) {
                weprintf("can't open %s:", argv[i]);
                continue;
            }
            if (grep(argv[1], f, argc>3 ? argv[i] : NULL) > 0)
                nmatch++;
            fclose(f);
        }
    }
    return nmatch == 0;
}
```

Обычно программы на С возвращают нуль в случае успеха и ненулевое значение в случае неудачи. Для нашей программы grep, как и для ее стандартной Unix-версии, успехом является нахождение совпадающих строк, поэтому она возвращает 0, если таковые имеются, 1, если нет, и 2 (через eprintf), если произошла ошибка. Эти значения можно проанализировать другой программой, например командным сценарием.

Функция grep считывает один файл, вызывая функцию match для каждой его строки:

```
/* grep: ищет выражение regexp в файле file */
int grep(char *regexp, FILE *f, char *name)
{
    int n, nmatch;
    char buf[BUFSIZ];
    nmatch = 0;
    while (fgets(buf, sizeof buf, f) != NULL) {
        n = strlen(buf);
        if (n > 0 && buf[n-1] == '\n')
            buf[n-1] = '\0';
        if (match(regexp, buf)) {
            nmatch++;
            if (name != NULL)
                printf("%s:", name);
            printf("%s\n", buf);
        }
    }
    return nmatch;
}
```

Главный модуль не завершает выполнение, если не удается открыть файл. Этот путь был выбран потому, что пользователи нередко пишут нечто вроде этого:

```
% grep herpolhode *.*
```

Затем оказывается, что один или несколько файлов в каталоге не открывается. Для grep лучше продолжить работу, сообщив об ошибке, чем сдаться и заставить пользователя набирать список файлов вручную. Заметьте также, что grep выводит

имя файла и соответствующую строку, однако подавляет вывод имени, если чтение выполняется из стандартного потока ввода или одного файла. Это может показаться странным, но на самом деле отражает устоявшуюся практику, обоснованную опытом. Если задан только один источник, задача grep обычно состоит в выборке строк, и имя файла было бы лишним среди результатов. А вот если программа разыскивает строки во многих файлах сразу, то задача чаще всего заключается в том, чтобы найти все вхождения определенной конструкции, и имена файлов являются ценной информацией. Сравните результаты двух команд:

```
% strings markov.exe | grep 'DOS mode'
```

и

```
% grep grammer chapter*.txt
```

Именно эти черты вместе со многими другими делают программу grep столь популярной и демонстрируют, что удачный выбор системы обозначений в сочетании с эргономическим подходом к делу позволяют создать естественный, удобный и мощный инструмент.

Наша реализация функции match возвращает управление, как только находит совпадение. Для программы grep это приемлемое правило, заданное по умолчанию. Однако для реализации, например, операции подстановки ("найти и заменить") в текстовом редакторе больше подходит алгоритм поиска *самого левого и самого длинного* совпадения. Например, в тексте "aaaaa" регулярное выражение a* дает совпадение с нулевой строкой уже в начале, однако более естественно было бы найти все пять букв a. Чтобы функция match находила самую левую и самую длинную строку-соответствие, функцию matchstar следовало бы переписать: вместо перебора всех символов слева направо она должна перепрыгивать через самую длинную строку, дающую совпадение с операндом звездочки, а затем возвращаться, если остаток строки не соответствует остатку заданного выражения. Другими словами, функция должна работать справа налево. Ниже приведена версия matchstar, выполняющая поиск самого левого и самого длинного соответствия.

```
/* matchstar: поиск по длинному левому совпадению c*regexp */
int matchstar(int c, char *regexp, char *text)
{
    char *t;
    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
    do { /* * нуль или больше соответствий */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}
```

Неважно, какое именно соответствие находит программа grep, поскольку она просто проверяет наличие такового и выводит всю строку. Раз функция с поиском самого длинного соответствия слева делает лишнюю работу, для программы grep она не нужна, хотя для операции поиска и замены является очень важной.

Наша реализация grep вполне конкурентоспособна по сравнению с системной версией, причем по любым поддерживаемым регулярным выражениям. Существуют случаи, вызывающие экспоненциальный рост затрат времени, например, поиск выражения a*a*a*a*a*b в тексте aaaaaaaaac. Однако экспоненциальная зависимость присутствует и в некоторых коммерческих реализациях. Один из вариантов grep в системе Unix под названием egrep основан на более изощренном алгоритме, который гарантирует линейную зависимость, избегая возврата назад, если частичное совпадение не дало результата.

Как же заставить функцию match обрабатывать любые регулярные выражения? Пришлось бы учесть классы символов наподобие [a-zA-Z] для поиска букв алфавита, возможность буквального употребления мета-символа (например, чтобы найти в тексте все точки), группировку с помощью скобок, выбор из альтернативных вариантов (abc или def). Первый шаг — это помочь функции match, скомпилировав образец в такое представление, которое было бы легче считывать. Слишком сложно и дорого анализировать класс символов каждый раз, когда берется очередной символ текста; более эффективный подход — использование предварительно обработанного представления, основанного на битовых векторах. Чтобы обеспечить поддержку полного языка регулярных выражений со скобками и альтернативным выбором, реализацию пришлось бы сделать намного сложнее. Но тем не менее и в ней можно использовать некоторые приемы, о которых пойдет речь далее в этой главе.

Упражнение 9.6. Сравните быстродействие функций match и strstr при анализе соответствия простой строки без мета-символов.

Упражнение 9.7. Напишите нерекурсивную версию функции matchhere и сравните ее быстродействие с рекурсивной версией.

Упражнение 9.8. Добавьте к grep некоторые ключи и опции. Среди самых популярных — ключи -v для обращения смысла поиска, -i для игнорирования регистра алфавитных символов и -n для включения в выходные данные номеров строк. Как должны выводиться номера строк? Следует ли включать их в те же строки, что и найденный текст?

Упражнение 9.9. Добавьте в функцию match поддержку операций + (одно или больше соответствий) и ? (нуль или больше соответствий). Так, выражение a+bb? соответствует строкам из одного или более a, за которым следует ни одного или больше b.

Упражнение 9.10. В текущей реализации функции match отключено специальное значение символов ^ и $, если выражение не начинается с них или не заканчивается ими, а также символа *, если за ним немедленно не следует литеральный символ или точка. Более привычный способ — это делать мета-символ буквальным, ставя перед ним обратную косую черту. Внесите исправления в функцию match, чтобы она именно так воспринимала обратные косые черты.

Упражнение 9.11. Добавьте в функцию match анализ классов символов. Класс символов дает совпадение, если найден хотя бы один из символов в квадратных скобках. Можно поступить еще удобнее, употребляя диапазоны, например, [a-z]

для нижнего регистра латинского алфавита или `[^0-9]` (с отрицанием) для поиска всех нецифровых символов.

Упражнение 9.12. Измените функцию `match` так, чтобы она использовала версию `matchstar` с поиском длинных левых соответствий, а затем модифицируйте ее так, чтобы возвращались позиции символов в начале и в конце найденного текстового совпадения. С ее помощью напишите программу `gres`, аналогичную `grep`, но выводящую все исходные строки после подстановки нового текста вместо того, который соответствует заданному образцу, например:

```
% gres 'homoiousian' 'homoousian' mission.stmt
```

Упражнение 9.13. Измените модули `match` и `grep` так, чтобы они работали со строками в кодировке UTF-8, состоящими из символов Unicode. Поскольку UTF-8 и Unicode являются надмножествами кодировки ASCII, это изменение поддерживает прямую (не обратную) совместимость. Как регулярные выражения, так и текст необходимо приспособить к кодировке UTF-8. Как в этом случае реализовать анализ классов символов?

Упражнение 9.14. Напишите программу автоматического тестирования регулярных выражений, которая бы генерировала тестовые выражения и тестовые строки для поиска соответствий. Если это возможно, воспользуйтесь существующей библиотекой, — возможно, и в ней найдутся ошибки.

9.3. Программирование в командных оболочках

Многие программы и утилиты структурно основаны на специализированном языке. Программа `grep` — это всего лишь один пример из большого семейства программ и оболочек, использующих регулярные выражения или другие языки для решения задач программирования.

Одним из первых подобных примеров был командный интерпретатор, или язык управления заданиями. Довольно рано возникла идея поместить часто используемые последовательности команд в файлы и выполнить экземпляр командного интерпретатора или оболочки с этим файлом в качестве входного. А оттуда уже оставался один шаг к добавлению параметров, условных конструкций, циклов, переменных и всех прочих прелестей обычных языков программирования. Основная разница состояла в том, что тип данных был только один — строковый, и отдельные операторы в таких программах сами представляли собой вызовы программ, выполняющих интересные и сложные вычисления. Хотя программирование в командных оболочках вышло из моды, уступив таким альтернативам, как Perl в системах с командной строкой или нажимание кнопок в графических средах, оно все еще остается эффективным способом сборки сложных утилит из более простых операций.

Одной из программируемых командных оболочек со своим языком является Awk. Этот небольшой специализированный язык основан на обработке текстовых шаблонов. Его основная задача — выбор и преобразование данных из входного пото-

ка. Awk автоматически считывает входные файлы и разбивает каждую строку на поля под именами от `$1` до `$NF`, где `NF` — количество полей в строке (см. главу 3). Предоставляя стандартные, отработанные средства для многих задач, этот язык позволяет быстро писать полезные подручные утилиты. Рассмотрим следующую законченную программу на Awk:

```
# split.awk: вывести поток по одному слову в строке
{ for (i = 1; i <= NF; i++) print $i }
```

Она выводит "слова" каждой из входных строк по одному на строке экрана. Ниже приведена реализация утилиты `fmt`, которая заполняет строки выходного потока словами так, чтобы длина их не превышала 60 символов. Пустая строка соответствует разрыву абзаца.

```
# fmt.awk: переформатирование в 60-символьные строки
/./ { for (i = 1; i <= NF; i++) addword($i) } # nonblank line
/^$/ { printline(); print "" } # blank line
END { printline() }
function addword(w) {
    if (length(line) + 1 + length(w) > 60)
        printline()
    if (length(line) == 0)
        line = w
    else
        line = line " " w
}
function printline() {
    if (length(line) > 0) {
        print line
        line = ""
    }
}
```

Мы часто пользуемся программой `fmt` для переформатирования почтовых сообщений и других коротких документов. Мы также пользовались ею для форматирования выходных потоков программ `markov` (см. главу 3).

Такие программируемые оболочки часто начинаются с мини-языков, разработанных для естественного выражения технических решений в узкоспециализированной области. Один из удачных примеров — это утилита `eqn` для Unix, используемая для набора математических формул. Ее входной язык почти совпадает с выражениями математиков при чтении формул вслух; например, $\frac{\pi}{2}$ записывается как `pi over 2`. В системе TEX используется тот же принцип; в ней эта формула записывается в виде `\pi \over 2`. Если для решаемой вами задачи существует естественная или близкая система обозначений и выражений, используйте или адаптируйте ее, — это лучше, чем начинать с нуля.

Толчком к созданию Awk послужила программа, распознававшая аномальные данные в журналах-протоколах телефонных переговоров с помощью регулярных выражений. Однако в Awk были включены переменные, выражения, циклы и т.д., что сделало его настоящим языком программирования. Perl и Tcl с самого начала разрабатывались так, чтобы сочетать удобство и выразительность мини-языков с

мощью настоящих языков программирования. Это действительно языки самого общего назначения, хотя чаще всего они используются для задач обработки текста.

Обычно в отношении языков таких программируемых оболочек употребляют термин *языки разработки сценариев* (*scripting languages*), поскольку они эволюционировали из более ранних командных интерпретаторов, программируемость которых ограничивалась выполнением последовательностей, или "сценариев", состоящих из запусков разных программ. Языки разработки сценариев допускают самые разные творческие применения регулярных выражений — не только поиск соответствий (распознавание наличия того или иного выражения в тексте), но также определение того, какие области текста подвергать определенной обработке. Именно это выполняется в двух командах `regsub` (сокращение от *regular expression substitution* — "подстановка регулярных выражений") в приведенной ниже программе на языке Tcl. Это некоторое обобщение программы, продемонстрированной нами в главе 4, которая извлекала информацию о биржевых котировках; теперь же наша программа извлекает и доставляет документ с URL-адресом, указанным в первом аргументе. Первая подстановка удаляет строку `http://`, если она присутствует; вторая заменяет первый символ / пустым местом, таким образом разбивая аргумент на два поля. Команда `lindex` извлекает из строки поля (начиная с индекса `0`). Текст, заключенный в квадратные скобки, выполняется как команда Tcl и заменяется получившимся в результате текстом; запись `$x` заменяется значением переменной `x`.

```
    # geturl.tcl: извлекает документ с указанным URL-адресом
    # входные данные: [http://]abc.def.com[/адрес...]
    regsub "http://" $argv "" argv  ;# удалить http:// если есть
    regsub "/" $argv " " argv       ;# заменить / пустым местом
    set so [socket [lindex $argv 0] 80] ;# открыть соединение
    set q "/[lindex $argv 1]"
    puts $so "GET $q HTTP/1.0\n\n"  ;# послать запрос
    flush $so
    while {[gets $so line] >= 0 && $line != ""} {} ;# пропустить
заголовок
    puts [read $so] ;# считать и вывести ответ
```

Обычно этот сценарий выдает большой объем выходных данных, значительная часть которого — теги HTML в угловых скобках (< >). Perl хорошо справляется с подстановкой в текстах, поэтому следующая наша утилита написана на этом языке с использованием регулярных выражений и подстановок, позволяющих избавиться от тегов:

```
    # unhtml.pl: удаляет теги HTML
    while (<>) {     # сбор данных в одну строку
        $str .= $_; # сцеплением входных строк
    }
    $str =~ s/<[^>]*>//g; # удаление <...>
    $str =~ s/ / /g; # замена   пустым местом
    $str =~ s/\s+/\n/g;   # сжатие пустого пространства
    print $str;
```

Этот пример невозможно понять, если не владеть языком Perl. Следующая конструкция подставляет строку `repl` вместо текста в `str`, соответствующего (по принципу левого самого длинного) регулярному выражению `regexp`:

```
    $str =~ s/regexp/repl/g
```

Последнее `g` (сокращение от *global*) означает, что это нужно сделать для всех соответствий в строке, а не только для первого. Мета-символ `\s` — это сокращение для произвольного символа пустого пространства (пробела, табуляции, символа новой строки и т.п.); `\n` обозначает конец строки. Строка `" "` является символом HTML (как те, что приведены в главе 2) и определяет неразрывный пробел.

Соберем все это вместе и продемонстрируем нелепый, однако работающий Web-браузер, реализованный в виде однострочного командного сценария:

```
# Web: получает Web-страницу и выводит текст, игнорируя HTML
geturl.tcl $1 | unhtml.pl | fmt.awk
```

Этот сценарий извлекает Web-страницу, отбрасывает всю управляющую и форматную информацию, а затем переформатирует текст по своим собственным правилам. Это очень быстрый способ загрузить страницу текста с Web-сайта.

Обратите внимание, сколько разных языков мы привлекли для решения комплексной задачи (Tcl, Perl, Awk), причем каждый из них особенно хорошо подходил для конкретной подзадачи. Регулярные выражения использовались в каждом из них. Секрет силы систем обозначений состоит как раз в том, чтобы найти удачную систему для каждой задачи. Язык Tcl особенно хорош для загрузки текстовой информации по сети; Perl и Awk удобно использовать для редактирования и форматирования текста, а регулярные выражения применяются для того, чтобы выбрать фрагменты текста для поиска и модификации. Все вместе эти языки гораздо мощнее, чем каждый из них по отдельности. Всегда целесообразно разбить целую задачу на части, если благодаря этому можно воспользоваться удачными системами обозначений для каждой части.

9.4. Интерпретаторы, компиляторы и виртуальные машины

Как программа проходит путь от исходного текста к исполняемому коду? Если язык достаточно прост (например, строка формата `printf` или простейшие регулярные выражения), можно выполнить программу прямо на ходу из исходного кода. Это легко и не требует промежуточных этапов.

Существует взаимно обратная зависимость между временем подготовки к выполнению и временем самого выполнения. Если язык сложен, то желательно преобразовать исходный текст в удобное и эффективное внутреннее представление для последующего выполнения. На предварительную обработку исходного текста уходит некоторое время, но оно окупается более быстрым выполнением. Программы, объединяющие преобразование и выполнение в один процесс, т.е. считывающие исходный текст, переводящие его на внутренний язык и сразу выполняющие, называются *интерпретаторами*. Языки Awk и Perl — интерпретируемые, как и многие другие языки для разработки сценариев или узкой специализации.

Еще один вариант — это сгенерировать инструкции для определенного вида системы, в которой программа предназначена работать. Это делают *компиляторы*. Такой процесс требует самой длительной предварительной подготовки, но зато дает самое быстрое выполнение в будущем.

Существуют и другие комбинации. Одну из них мы рассмотрим в этом разделе — это компиляция программы в набор инструкций искусственной системы (*виртуальной машины*), которую можно имитировать на любом реальном компьютере. Виртуальная машина сочетает в себе многие преимущества как традиционной компиляции, так и интерпретирования.

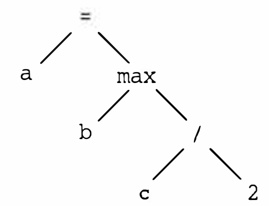
Если язык достаточно прост, то выделить в программе структуру и преобразовать ее во внутреннюю форму достаточно просто и не отнимает много времени. Но если язык содержит сложные средства — декларации, вложенные управляющие структуры, рекурсивно определенные операторы или выражения, операции с разным приоритетом и т.п., — то синтаксический анализ кода и определение структуры кода становится нелегкой задачей.

Синтаксические анализаторы часто пишутся с помощью автоматических генераторов, еще называемых *компиляторами компиляторов*, например `yacc` или `bison`. Такие программы преобразуют описание языка (его *грамматику*) в программу на C (как правило) или на C++, которая после компиляции уже может транслировать операторы языка во внутреннее представление. Генерирование синтаксических анализаторов непосредственно из грамматик — это еще одна демонстрация эффективности хорошей системы обозначений.

Представление, создаваемое синтаксическим анализатором, обычно является деревом, внутренние узлы которого содержат операции, а листья — операнды. Например, после разбора оператора

```
a = max(b, c/2);
```

возникает следующее синтаксическое дерево:



Многие из алгоритмов работы с деревьями, описанные в главе 2, могут использоваться для построения и разбора синтаксических деревьев.

После того как дерево построено, его обработку можно продолжить разными способами. Самый прямой, используемый в Awk, состоит в том, чтобы непосредственно обойти дерево, выполняя вычисления в узлах по мере обхода. Упрощенную версию процедуры вычислений для условного языка целочисленных выражений можно написать, например, с применением концевого обхода:

```
typedef struct Symbol Symbol;
typedef struct Tree Tree;

struct Symbol {
    int value;
    char *name;
};

struct Tree {
```

```
    int op;          /* код операции */
    int value;       /* значение, если число */
    Symbol *symbol; /* Symbol, если переменная */
    Tree *left;
    Tree *right;
};
/* eval: версия 1: вычисление выражения на дереве */
int eval(Tree *t)
{
    int left, right;
    switch (t->op) {
    case NUMBER:
        return t->value;
    case VARIABLE:
        return t->symbol->value;
    case ADD:
        return eval(t->left) + eval(t->right);
    case DIVIDE:
        left = eval(t->left);
        right = eval(t->right);
        if (right == 0)
            eprintf("divide %d by zero", left);
        return left / right;
    case MAX:
        left = eval(t->left);
        right = eval(t->right);
        return left>right ? left : right;
    case ASSIGN:
        t->left->symbol->value = eval(t->right);
        return t->left->symbol->value;
    /* ... */
    }
}
```

В первых блоках `case` вычисляются простые выражения типа констант и значений, далее — арифметические выражения, а еще дальше рассматриваются специальные случаи, условные конструкции и циклы. Для реализации управляющих структур на дереве необходимо было бы разместить дополнительную информацию, которая здесь не показана.

Как и в функциях `pack` и `unpack`, можно заменить явный оператор `switch` таблицей указателей на функции. Отдельные операции будут примерно такими же, как и в операторе `switch`:

```
/* addop: возвращает сумму двух выражений на дереве */
int addop(Tree *t)
{
    return eval(t->left) + eval(t->right);
}
```

Таблица указателей на функции сопоставляет операции с функциями, которые их выполняют:

```
enum { /* коды операций, Tree.op */
    NUMBER,
    VARIABLE,
```

```
        ADD,
        DIVIDE,
        /* ... */
    };
    /* optab: таблица функций для операций */
    int (*optab[])(Tree *) = {
        pushop, /* NUMBER */
        pushsymop, /* VARIABLE */
        addop, /* ADD */
        divop, /* DIVIDE */
        /* ... */
    };
```

При вычислении по знаку операции в таблице разыскивается указатель на функцию, а затем эта функция вызывается. В данной версии рекурсивно вызываются и другие функции.

```
    /* eval: версия 2: вычисления на дереве по таблице операций */
    int eval(Tree *t)
    {
        return (*optab[t->op])(t);
    }
```

Обе версии функции `eval` — рекурсивные. Существуют способы избежать рекурсии, в том числе искусный прием организации *многопоточного кода*, который делает стек вызовов совсем мелким. Самый аккуратный метод избавления от рекурсии — это помещение функций в массив и его последовательный перебор с целью выполнения программы. Этот массив фактически является последовательностью инструкций, выполняемых небольшой специализированной виртуальной машиной.

И все-таки для представления частично вычисленных значений нам нужен стек. Форма функции изменяется, но это изменение легко проследить. Фактически мы изобрели *стековую машину*, в которой инструкции являются маленькими функциями, а операнды помещаются в отдельный стек операндов. Это не настоящая машина, но программировать ее можно вполне реально, и она легко поддается реализации в виде интерпретатора.

Вместо обхода дерева с целью вычислений на нем можно обходить его для генерирования массива функций, предназначенных для выполнения программы. Такой массив будет также содержать значения данных, используемых в инструкциях (например, констант и переменных-символов), так что тип элементов в массиве должен быть объединением (`union`):

```
    typedef union Code Code;
    union Code {
        void (*op)(void); /* функция, если знак операции */
        int value;        /* значение, если число */
        Symbol *symbol;   /* Symbol, если переменная */
    };
```

Ниже приведена функция для генерирования указателей на функции и помещения их в массив `code` соответствующих объектов. Возвращаемое из `generate` значение является не значением выражения — оно будет вычислено только при выполнении сгенерированного кода, — а индексом следующей генерируемой операции в массиве `code`:

```
/* generate: генерирует инструкции при обходе дерева */
int generate(int codep, Tree *t)
{
    switch (t->op) {
    case NUMBER:
        code[codep++].op = pushop;
        code[codep++].value = t->value;
        return codep;
    case VARIABLE:
        code[codep++].op = pushsymop;
        code[codep++].symbol = t->symbol;
        return codep;
    case ADD:
        codep = generate(codep, t->left);
        codep = generate(codep, t->right);
        code[codep++].op = addop;
        return codep;
    case DIVIDE:
        codep = generate(codep, t->left);
        codep = generate(codep, t->right);
        code[codep++].op = divop;
        return codep;
    case MAX:
        /* ... */
    }
}
```

Для оператора `a = max(b, c/2)` сгенерированный код выглядел бы следующим образом:

```
pushsymop
b
pushsymop
c
pushop
2
divop
maxop
storesymop
a
```

Функции для реализации операций манипулируют стеком, извлекая операнды и помещая результаты.

Интерпретатор представляет собой цикл с перебором счетчика инструкций по массиву указателей на функции:

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* счетчик инструкций */

/* eval: версия 3: вычисляет выражение по сгенерированному коду
*/
int eval(Tree *t)
{
    pc = generate(0, t);
    code[pc].op = NULL;
```

```
        stackp = 0;
        pc = 0;
        while (code[pc].op != NULL)
            (*code[pc++].op)();
        return stack[0];
    }
```

В этом цикле выполняется программная эмуляция стековой машины, которая по сути делает то, что происходит в реальном компьютере на аппаратном уровне. Вот несколько характерных операций:

```
    /* pushop: помещает число в стек; число - следующее слово в
потоке кода */
    void pushop(void)
    {
        stack[stackp++] = code[pc++].value;
    }

    /* divop: вычисляет отношение двух выражений */
    void divop(void)
    {
        int left, right;
        right = stack[--stackp];
        left = stack[--stackp];
        if (right == 0)
            eprintf("divide %d by zero\n", left);
        stack[stackp++] = left / right;
    }
```

Обратите внимание, что проверка деления на нуль фигурирует в функции `divop`, а не в `generate`.

Условное выполнение, ветвление и циклы выполняются путем модифицирования счетчика инструкций. В результате управление передается в другое место массива функций. Например, оператор `goto` всегда устанавливает значение переменной `pc`, а условный оператор — только если условие истинно.

Естественно, что массив `code` является внутренним для интерпретатора. Но представим себе, что сгенерированную программу необходимо сохранить в файле. Если выписать явные адреса функций, получится непереносимый и неустойчивый результат. Но вместо этого можно выписать константы, представляющие эти функции, — скажем, `1000` для `addop`, `1001` для `pushop` и т.д., — а затем при считывании программы в память для выполнения преобразовать эти относительные константы снова в указатели на функции.

Если взглянуть на файл, генерируемый этой процедурой, он напомнит нам поток инструкций виртуальной машины, реализующих элементарные операции нашего мини-языка, а функция `generate` окажется компилятором языка в коды виртуальной машины. Вообще, виртуальная машина — идея не новая, но недавно она снова вошла в моду благодаря языку Java и его виртуальной машине Java Virtual Machine (JVM). С их помощью легко создавать хорошо переносимые, высокоэффективные представления программ, написанных на языке высокого уровня.

9.5. Программы, пишущие программы

Возможно, наиболее примечательная черта функции `generate` состоит в том, что это программа, пишущая другую программу: ее выходные данные представляют собой поток выполняемых инструкций для другой (виртуальной) системы или машины. Это именно то, что делают компиляторы, переводящие исходный код на язык машинных инструкций, так что идея, конечно, не нова. На самом деле программы, пишущие другие программы, попадаются то тут, то там в самых разных формах.

Один из распространенных примеров — это динамическое генерирование HTML-кода для Web-страниц. HTML — это язык, хотя и ограниченный, причем в своем составе он может содержать код JavaScript. Web-страницы часто генерируются в реальном времени (на лету) программами, написанными на языках Perl или C, и содержат специфическую информацию (например, результаты поиска или целенаправленную рекламу), определяемую поступающим на сервер запросом. В этой книге мы пользовались специализированными языками для описания графиков, рисунков, таблиц, математических выражений и предметного указателя. Еще один пример — это PostScript, язык программирования, текст на котором генерируется текстовыми редакторами, графическими и многими другими программами. На финальном этапе допечатной подготовки вся эта книга была представлена в виде PostScript-программы из 57 000 строк.

Документ — это, конечно, статическая программа, но тем не менее идея использования языка программирования как системы обозначений в специализированной области знаний остается исключительно плодотворной. Много лет назад программисты мечтали, чтобы компьютеры писали все программы за них самостоятельно. Возможно, эта мечта никогда не осуществится, однако современные компьютеры уже пишут довольно много программ за нас, причем часто для представления такой информации, которую мы раньше и не подумали бы описывать на языках программирования.

Наиболее распространенная программа, пишущая другие программы, — это компилятор, переводящий с языка высокого уровня на язык машинных инструкций. Но часто бывает полезно транслировать исходный код в код на другом языке, также высокого уровня. В предыдущем разделе упоминались генераторы синтаксических анализаторов, которые преобразуют определения грамматик языков в программы на C, выполняющие синтаксический разбор этих языков. Для этих целей часто используется язык C, который в этом случае служит чем-то вроде "языка ассемблера высокого уровня". Modula-3 и C++ как раз являются примерами языков общего назначения, первые компиляторы которых создавали C-код, транслируемый затем стандартным компилятором C. Этот подход имеет ряд преимуществ — в том числе преимущества быстродействия (поскольку программы выполняются со скоростью C-программ) и переносимости (компиляторы можно перенести в любую среду, в которой имеется компилятор C). На раннем этапе существования упомянутых языков это очень помогло их всеобщему распространению.

Еще один пример можно привести в связи с Visual Basic. Его графический интерфейс генерирует набор операторов присваивания этого языка для инициализации объектов, которые пользователь выбирает из меню и позиционирует на экране с помощью мыши. Многие другие языки также имеют "визуальные" среды програм-

мирования и программы-“мастера”, которые синтезируют код пользовательских интерфейсов по нескольким щелчкам мышью.

Несмотря на богатые возможности генераторов программ, так же как и на существование множества других удачных примеров, специальные системы обозначений ценятся не так высоко, как следовало бы, и отдельные программисты пользуются ими нечасто. А ведь всегда есть много возможностей создать некоторую часть кода с помощью программы, воспользовавшись преимуществами автоматизации. Далее приведено несколько примеров с генерированием кода на языках C и C++.

Операционная система Plan 9 генерирует сообщения об ошибках на основании заголовочного файла, содержащего имена и комментарии. Комментарии автоматически конвертируются в литеральные строки в кавычках и объединяются в массив, индексируемый перечислимыми символическими константами. Вот структура заголовочного файла:

```
/* errors.h: стандартные сообщения об ошибках */
enum {
    Eperm,  /* Отказано в допуске */
    Eio,    /* Ошибка ввода-вывода */
    Efile,  /* Файл не существует */
    Emem,   /* Исчерпан лимит памяти */
    Espace, /* Не хватает места в файле */
    Egreg   /* Виноват Гриша */
};
```

Получив на вход эти данные, несложная программа генерирует следующий набор деклараций для сообщений об ошибках:

```
/* сгенерировано автоматически; не править. */
char *errs[] = {
    "Отказано в допуске", /* Eperm */
    "Ошибка ввода-вывода", /* Eio */
    "Файл не существует", /* Efile */
    "Исчерпан лимит памяти", /* Emem */
    "Не хватает места в файле", /* Espace */
    "Виноват Гриша", /* Egreg */
};
```

Этот подход имеет ряд преимуществ. Во-первых, связь между перечислимыми константами и строками, которые они представляют, самодокументирована и строго формальна. Информация фигурирует только в одном месте, и из этого одного места генерируется весь код, поэтому обновлять данные достаточно легко. Если бы информация была разбросана, неизбежно возникала бы ее рассинхронизация. Наконец, легко устроить так, чтобы соответствующий C-файл генерировался и компилировался заново всякий раз, когда в заголовочный файл вносятся изменения. Если необходимо изменить текст какого-либо сообщения об ошибке, достаточно модифицировать заголовочный файл и перекомпилировать систему. Все сообщения во всех местах обновятся автоматически.

Программу-генератор можно написать на любом языке. Это легко сделать, например, на языке обработки строк типа Perl:

```
# enum.pl: генерирует сообщения об ошибках из enum и комментариев
print "/* сгенерировано автоматически; не править. */\n\n";
print "char *errs[] = {\n";
while (<>) {
    chop;                          # удаление конца строки
    if (/^\s*(E[a-z0-9]+),?/) {    # первое слово - E...
        $name = $1;                # сохранение имени
        s/.*\/\* *//;              # удаление вплоть до /*
        s/ *\*\///;                # удаление */
        print "\t\"$_\", /* $name */\n";
    }
}
print "};\n";
```

Здесь снова работают регулярные выражения. Выбираются строки, первые поля которых являются идентификаторами с запятыми после них. При первой подстановке удаляются все символы до первого непустого символа комментария, а при второй — закрывающая скобка комментария и пустые символы перед ней.

В ходе тестирования одного из компиляторов Энди Кёниг (Andy Koenig) разработал удобный способ оформления кода на C++ для проверки того, что компилятор действительно способен находить ошибки в программе. Фрагменты кода, которые должны были вызывать диагностические сообщения компилятора, сопровождались специальными комментариями с описаниями ожидаемых сообщений. Каждая такая строка начиналась с `///` (чтобы отличаться от обычных комментариев) и регулярного выражения, соответствовавшего диагностическим сообщениям к этой строке. Например, следующие фрагменты кода должны привести к выдаче диагностики:

```
int f() {}
    /// warning.* non-void function .* should return a value

void g() {return 1;}
    /// error.* void function may not return a value
```

Если пропустить второй фрагмент через компилятор C++, он выведет ожидаемое сообщение, соответствующее заданному регулярному выражению:

```
% CC x.c
"x.c", line 1: error(321): void function may not return a value
```

Каждый такой фрагмент кода компилируется, а затем выходные данные компиляции сравниваются с ожидаемой диагностикой. Этот процесс выполняется командным сценарием с привлечением программ на Awk. Если тест заканчивается неудачей, значит, сообщения компилятора отличаются от ожидаемых. Поскольку комментарии представляют собой регулярные выражения, сравнение производится по нестрогим правилам — несколько произвольно. Их можно сделать более или менее строгими в зависимости от цели тестирования.

Идея комментариев, имеющих определенное семантическое значение для программы, не нова. Они существуют в языке PostScript, где обычные комментарии начинаются со знака `%`, а те, которые начинаются с двойного знака `%%`, могут нести

дополнительную информацию о номерах страниц, ограничительных рамках страницы, именах шрифтов и т.д.:

```
%%PageBoundingBox: 126 307 492 768
%%Pages: 14
%%DocumentFonts: Helvetica Times-Italic Times-Roman
          LucidaSans-Typewriter
```

В языке Java комментарии, начинающиеся с `/**` и заканчивающиеся `*/`, используются для создания документации на класс, определение которого следует за комментарием. Обобщением самодокументированного кода является так называемое *содержательное программирование* (*literate programming*), при котором программа и ее документация интегрируются в одно целое, позволяющее с помощью одного процесса напечатать документацию для чтения, а с помощью другого — подготовить программу к компиляции и выполнению.

Во всех приведенных примерах важно отметить роль систем обозначений, смешения и совместного использования языков, активное привлечение утилит и оболочек. Все эти сочетания позволяют усилить эффект применения отдельных компонентов.

Упражнение 9.15. Одна из старых курьезных задач программирования состоит в том, чтобы написать такую программу, которая бы при выполнении выводила в точности сама себя в виде исходного кода. Это очень особый случай программы, пишущей программу. Попробуйте-ка сделать это на одном из ваших любимых языков.

9.6. Генерирование кода с помощью макросов

Если задача не чрезмерно велика и писать программу для генерирования программы нецелесообразно, можно прибегнуть к помощи макросов, которые бы создавали часть кода на этапе компиляции. В этой книге мы неоднократно призывали не пользоваться макросами и условной компиляцией. Эти средства портят стиль программирования и создают проблемы. Но все-таки у них есть свое место под солнцем и право на существование. В конце концов, иногда именно простая текстовая подстановка дает правильное решение задачи. В нашем примере мы воспользуемся макропроцессором C/C++ для сборки одной искусственной программы, изобилующей повторениями.

Например, программа для оценки быстродействия элементарных языковых конструкций (см. главу 7) использует препроцессор для сборки тестов в единое целое путем их заключения в код-оболочку. Суть теста состоит в том, чтобы инкапсулировать фрагмент кода в цикл, в котором запускается таймер, фрагмент выполняется много раз, фиксируются показания таймера и выводится отчет о результатах. Весь повторяемый код включен в несколько макроопределений, а код, подвергающийся измерениям, передается в них как аргумент. Основной макрос приобретает следующую форму:

```
#define LOOP(CODE) {                        \
    t0 = clock();                           \
    for (i = 0; i < n; i++) { CODE; }       \
    printf("%7d ", clock() - t0);           \
}
```

Обратные косые черты позволяют распространить тело макроса на несколько строк. Этот макрос используется в "операторах", обычно выглядящих следующим образом:

```
LOOP(f1 = f2)
LOOP(f1 = f2 + f3)
LOOP(f1 = f2 - f3)
```

Имеется еще несколько операторов инициализации, однако основная измерительная часть представлена этими макровызовами с одним аргументом, которые раскрываются в довольно большой объем кода.

Макропроцессор можно использовать и для генерирования профессионального, коммерчески значимого кода. Барт Локанти (Bart Locanthi) однажды написал высокоэффективную версию двумерной графической операции. Эта операция, называющаяся `bitblt` или `rasterop`, трудно поддается оптимизации быстродействия, потому что в ней много аргументов комбинируется сложным способом. Однако после тщательного анализа вариантов Локанти развел эти комбинации по разным циклам, которые можно было оптимизировать независимо. Каждый случай затем кодировался с применением макроподстановки, аналогичной примеру с тестированием быстродействия; все варианты были учтены в одном большом операторе `switch`. Первоначальный исходный код имел длину несколько сотен строк, а после развертывания макросов приобретал длину в несколько тысяч. Развернутый таким образом код не был оптимальным, однако, учитывая сложность задачи, вполне практичным и простым в разработке. К тому же для высокоэффективного кода он был достаточно хорошо переносимым.

Упражнение 9.16. В упражнении 7.7 требуется написать программу для измерения затрат на различные операции в C++. Воспользуйтесь идеями этой главы и напишите другую версию этой программы.

Упражнение 9.17. В упражнении 7.8 требуется разработать модель стоимости для языка Java, в котором макроопределения не предусмотрены. Решите эту проблему, написав другую программу на каком-нибудь другом языке (или языках) по своему выбору, которая бы сама генерировала Java-версию и автоматизировала измерение времени.

9.7. Компиляция в реальном времени

В предыдущем разделе мы рассматривали программы, которые пишут программы. В каждом из примеров сгенерированная программа возникала в виде исходного кода, и ее еще нужно было компилировать или интерпретировать для последующего выполнения. Однако можно генерировать и код, сразу же готовый к выполнению. Для этого генерируется не исходный код, а поток машинных инструкций. Этот процесс известен под названием компиляции "в реальном времени" (on-the-fly) или "по требованию" (just-in-time). С учетом удобного и звучного сокращения JIT, второй термин несколько популярнее.

Скомпилированный код неизбежно непереносим — он работает только на одном типе процессоров, хотя и очень быстро. Рассмотрим следующее выражение:

```
max(b, c/2)
```

Необходимо получить значение c, разделить на два, сравнить результат с b и выбрать большее. Если вычислять выражение на той виртуальной машине, которую мы обрисовали в общих чертах ранее, можно было бы устранить проверку деления на нуль в divop. Поскольку 2 никогда не равно 0, эта проверка все равно бесполезна. Однако при любом варианте реализации виртуальной машины, который мы наметили, устранить эту проверку никак не получится. При выполнении операции деления делитель неизменно сравнивается с нулем, как бы эта операция ни была реализована. Вот где может помочь динамическое генерирование кода. Если построить код для выражения непосредственно, а не просто собрать в кучу заранее определенные операции, можно избежать проверки делителя, который и так гарантированно не равен нулю. На самом деле можно зайти даже дальше. Если все выражение постоянно, как, например, max(3*3, 4/2), можно вычислить его один раз при генерировании кода и заменить константой 9. Если это выражение фигурирует в цикле, то при каждом его проходе будет экономиться некоторое время. Если цикл выполняется достаточно много раз, то экономия от его оптимизации оправдает время, затраченное на анализ выражения и генерирование оптимального кода для него.

Ключевая идея состоит в том, что система обозначений дает нам общий способ описания задачи, однако компилятор для этой системы может еще и приспосабливать код к специфическим деталям вычислений. Например, в виртуальной машине для регулярных выражений полезно было бы иметь операцию для сравнения литеральных символов:

```
int matchchar(int literal, char *text)
{
    return *text == literal;
}
```

Однако если мы генерируем код для конкретного заданного текста, то значение того или иного литерала уже фиксировано — например, 'x' — и вместо этого можно использовать такую операцию:

```
int matchx(char *text)
{
    return *text == 'x';
}
```

Вместо того чтобы заранее определять специальную операцию для каждого литерального символьного значения, можно упростить дело и генерировать код оперативно только для тех операций, которые нам действительно нужны для обработки текущего выражения. Обобщая идею на полный набор операций, можно написать компилятор реального времени для трансляции регулярных выражений в специальный код, оптимизированный для этих конкретных выражений.

Именно это сделал Кен Томпсон (Ken Thompson) при реализации регулярных выражений на машине IBM 7094 в 1967 г. Его версия генерировала небольшие блоки двоичных инструкций IBM 7094 для различных операций выражения, затем

"сшивала" их вместе и запускала получившуюся программу путем ее вызова как обычной функции. Аналогичные методы можно применять для создания специфических потоков инструкций, обновляющих экран в графических системах, где существует так много особых случаев и исключений, что эффективнее генерировать динамический код для каждого из них на ходу, чем выписать их все заранее или включить условное ветвление в код более общего характера.

Демонстрация реального компилятора "по требованию" увела бы нас в дебри конкретного набора машинных инструкций, но тем не менее посмотреть хотя бы в общих чертах, как работают подобные системы, очень полезно. Остальная часть этого раздела посвящена скорее общим идеям и рассуждениям, чем деталям конкретных реализаций.

Вспомним, что мы оставили нашу виртуальную машину в следующем виде:

```
Code code[NCODE];
int stack[NSTACK];
int stackp;
int pc; /* счетчик инструкций */
    ...
    Tree *t;

    t = parse();
    pc = generate(0, t);
    code[pc].op = NULL;

    stackp = 0;
    pc = 0;
    while (code[pc].op != NULL)
        (*code[pc++].op)();
    return stack[0];
```

Чтобы приспособить этот код к компиляции в реальном времени, необходимо внести некоторые изменения. Во-первых, массив code должен стать массивом не указателей на функции, а выполняемых инструкций. Будут ли инструкции иметь тип char, int или long, зависит от конкретного процессора, для которого выполняется компиляция. Пока предположим тип int. После генерирования кода он будет вызываться как функция. Виртуального счетчика инструкций здесь быть не должно, поскольку код будет последовательно перебираться в собственном цикле процессора. После окончания вычислений произойдет возврат управления, как из обычной функции.

Также мы можем выбрать между ведением отдельного стека операндов для нашей машины и использованием собственного стека процессора. Каждый из подходов имеет свои преимущества, но мы решили обойтись отдельным стеком и сосредоточиться на подробностях самого кода. В этом случае реализация машины приобретает следующий вид:

```
typedef int Code;
Code code[NCODE];
int codep;
int stack[NSTACK];
int stackp;
    ...
    Tree *t;
```

```
    void (*fn)(void);
    int pc;
    t = parse();
    pc = generate(0, t);
    genreturn(pc); /* генерирование кода для возвращения */
    stackp = 0;
    flushcaches(); /* синхронизация памяти с процессором */
    fn = (void(*)(void)) code; /* приведение массива к указателю */
    (*fn)();        /* вызов функции */
    return stack[0];
```

После завершения функции generate функция genreturn создает последовательность инструкций для возвращения управления из сгенерированного кода в функцию eval.

Функция flushcaches реализует операции по подготовке процессора к выполнению сгенерированного кода. Современные компьютеры и системы работают быстро в частности потому, что хранят инструкции и данные в кэш-памяти и используют внутренние конвейерные каналы, позволяющие параллельно выполнять несколько последовательных инструкций. Кэш-память и конвейеры воспринимают поток инструкций как статический; если же код сгенерирован непосредственно перед выполнением, процессор может запутаться. Процессору необходимо сбросить все из конвейерного канала и очистить кэш-память, а уже потом выполнять новые инструкции. Все эти операции целиком и полностью аппаратно-зависимы. Реализации функции flushcaches на разных компьютерах будут совершенно различны.

Примечательное выражение (void(*)(void))code служит для преобразования адреса массива, который содержит сгенерированные инструкции, в указатель на функцию, который можно использовать для вызова этого кода как функции.

Генерирование кода не составляет особых технических трудностей, хотя для того, чтобы делать это эффективно, придется потрудиться. Начнем с некоторых "кирпичиков". Как и раньше, в процессе компиляции запоминаются массив code и его индекс. Для простоты мы сделали их глобальными, как и раньше. Теперь можно написать функцию для помещения инструкций в код:

```
/* emit: добавление инструкции в поток кода */
void emit(Code inst)
{
    code[codep++] = inst;
}
```

Сами инструкции могут определяться в виде процессорно-зависимых макросов или мини-функций, которые "собирают" инструкции путем заполнения полей слова инструкции. Гипотетически можно было бы написать функцию popreg для генерирования кода по извлечению числа из стека и помещению его в регистр процессора и еще одну, pushreg, для генерирования кода по извлечению числа из регистра и помещения в стек. Наша новая функция addop пользовалась бы ими следующим образом, при наличии предопределенных констант, описывающих инструкции (типа ADDINST) и их положение (различные сдвиги SHIFT, определяющие формат):

```
/* addop: генерирует инструкцию ADD */
void addop(void)
{
```

```
        Code inst;
        popreg(2);    /* извлечь из стека в регистр 2 */
        popreg(1);    /* извлечь из стека в регистр 1 */
        inst = ADDINST << INSTSHIFT;
        inst |= (R1) << OP1SHIFT;
        inst |= (R2) << OP2SHIFT;
        emit(inst);   /* выдать ADD R1, R2 в поток */
        pushreg(2);   /* поместить значение регистра 2 в стек */
    }
```

Все это лишь начало. Если бы пришлось действительно писать компилятор реального времени, необходимо было бы прибегнуть к оптимизации. Для добавления константы не нужно помещать ее в стек, извлекать оттуда, а затем уже добавлять; достаточно добавить ее непосредственно. Аналогичные соображения позволяют оптимизировать почти все нерациональные операции. Но даже в этом виде функция `addop` уже работает значительно быстрее, чем более ранние версии, потому что разные операции не сшиваются вместе посредством вызовов отдельных функций. Вместо этого код для их выполнения помещается в память в виде единого блока инструкций, а их перебор выполняется реальным счетчиком процессора.

Функция `generate` выглядит практически так же, как и в реализации виртуальной машины. Однако в этот раз она генерирует реальные инструкции этой машины, а не указатели на заранее определенные функции. Чтобы генерировать высокоэффективный код, эта функция должна приложить некоторые усилия для поиска подставляемых констант и выполнения других аналогичных оптимизаций.

Итак, мы вкратце рассмотрим вопросы генерирования кода, в общих чертах наметив некоторые методы работы реальных компиляторов и полностью проигнорировав многие другие. Также в нашем изложении пропущены многие проблемы, вызванные сложным устройством современных процессоров. И все же мы продемонстрировали на примерах, что программа может анализировать описание задачи и продуцировать специализированный код для ее эффективного решения. Этими идеями можно воспользоваться для самостоятельной разработки сверхбыстрой версии `grep`, реализации мини-языка для своих собственных целей, разработки и реализации виртуальной машины, оптимизированной для специальных вычислений, или даже, с некоторой помощью, написания компилятора для нового интересного языка.

Регулярное выражение имеет мало внешнего сходства с программой на языке C++, однако и то и другое — системы обозначений для решения определенных задач. Имея правильную систему обозначений, можно облегчить трудности решения многих проблем. Разработка и реализация системы обозначений — это необычное и увлекательное занятие.

Упражнение 9.18. Компилятор реального времени сгенерирует более быстрый код, если он сможет заменить выражения типа `max(3*3, 4/2)`, содержащие только константы, их явными значениями. Как компилятор должен вычислять значение такого выражения после того, как оно опознано им?

Упражнение 9.19. Как бы вы протестировали компилятор реального времени?

Дополнительная литература

Подробное обсуждение вопросов программирования с применением утилит и командных оболочек, которое столь развито в среде Unix, содержится в книге Brian Kernighan, Rob Pike, *The Unix Programming Environment* (Prentice Hall, 1984). Глава 8 этой книги представляет полную реализацию несложного языка программирования — от грамматики в формате `yacc` до исполняемого кода.

В книге Don Knuth, *TEX: The Program* (Addison-Wesley, 1986) описывается сложная программа форматирования документов; вся программа (около 13 000 строк на языке Pascal) представлена в стиле "содержательного программирования", в котором текст программы содержит подробные пояснения и служит сам себе документацией, потому что именно из него документация извлекается специальными программами. В книге Chris Fraser, David Hanson, *A Retargetable C Compiler: Design and Implementation* (Addison-Wesley, 1995) предлагается такого же рода описание одного из компиляторов ANSI C.

Виртуальная машина Java описана в книге Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification*, 2nd Edition (Addison-Wesley, 1999).

Алгоритм Кена Томпсона (Ken Thompson), один из первых запатентованных программных продуктов, представлен в статье "Regular Expression Search Algorithm", *Communications of the ACM*, 1968, **11**, 6, p. 419–422. Очень подробное изложение этого предмета можно найти в книге Jeffrey E. F. Friedl, *Mastering Regular Expressions* (O'Reilly, 1997).

Компилятор в реальном масштабе времени для операций двумерной графики описан в статье Rob Pike, Bart Locanthi, John Reiser, "Hardware/Software Tradeoffs for Bitmap Graphics on the Blit", *Software—Practice and Experience*, February 1985, **15**, 2, p. 131–152.

Эпилог

Если бы только люди могли учиться на примерах из истории, какие уроки мы могли бы получить! Но, увы, нас ослепляют страсти и пристрастия, а свет жизненного опыта подобен фонарю на корме судна, который освещает лишь волны позади нас!

Сэмюэл Тейлор Кольридж. "Воспоминания"
(Samuel Taylor Coleridge, Recollections*)*

Мир программирования и вычислительной техники постоянно изменяется, и интенсивность этих изменений все нарастает. Программистам приходится осваиваться с новыми языками, новыми утилитами, новыми системами и средами, а также с несовместимыми изменениями предыдущих версий. Программы разрастаются, интерфейсы усложняются, сроки сдачи работ сокращаются.

И все же существует нечто постоянное, некие скалы в бурном море — очаги стабильности, благодаря которым уроки прошлого можно применить в будущем. Эта книга во многом основана именно на этих вечных ценностях.

Простота и **ясность** — это первые и наиболее важные принципы, поскольку почти все остальные следуют из них. Реализуйте самый простой метод из тех, которые работают. Выбирайте самый простой алгоритм из всех достаточно быстрых для данной задачи и самую простую структуру данных, пригодную для решения. Соберите их в единое целое в простом и понятном коде. Не усложняйте без причины, если требования быстродействия не вынуждают вас к этому. Интерфейсы должны быть лаконичными и экономными — по крайней мере, до тех пор, пока не станет совершенно очевидно, что получаемые преимущества окупают их повышенную сложность.

Общность часто идет рука об руку с простотой, поскольку позволяет решить задачу раз и навсегда, а не решать ее снова и снова для частных случаев. Часто это правильный подход и к переносимости: найти единственное общее решение, работающее во всех системах, вместо того, чтобы только подчеркивать и усиливать различия между операционными средами.

Эволюционность развития — это следующий принцип. Невозможно создать идеальную программу с первого же раза. Интуиция, необходимая для нахождения правильного решения, приходит только с опытом глубоких размышлений над задачами и проблемами. Хорошую систему не удастся построить ни на одном кавалерийском натиске, ни на одной усидчивости. Нужно считаться с реакцией пользователей; наи-

более эффективным следует считать цикл, состоящий из разработки, экспериментирования, обратной связи от пользователя и дальнейшей доработки. Программы, которые мы пишем для себя, часто не получают никакого развития, а программы, покупаемые у других, нередко изменяются слишком быстро, но не обязательно в лучшую сторону.

Интерфейсы составляют значительную часть стратегии программирования; вопросы взаимодействия компонентов вычислительного процесса возникают во многих случаях. Самый распространенный случай — библиотеки, однако существуют еще интерфейсы между программами, а также между пользователями и программами. Принципы простоты и общности играют особенно важную роль именно в проектировании интерфейсов. Делайте их согласованными, простыми в изучении и использовании, а затем придерживайтесь их как можно строже. Эффективной техникой является абстрагирование: вообразите себе идеально работающий компонент, библиотеку или программу; затем подгоните интерфейс под этот идеал, насколько возможно; скройте подробности реализации за четкой непроницаемой границей.

Автоматизация — это принцип, который обычно несколько недооценивается. Часто бывает удобнее заставить компьютер сделать работу за вас, чем делать ее от руки. Мы видели примеры этого подхода в тестировании, в отладке, в анализе быстродействия, а также в немалой степени при написании кода, где при правильной формулировке задачи программы могут сами писать такие программы, которые было бы трудно написать людям.

Выбор системы обозначений также недооценивается, причем не только как способ сообщить компьютеру, что он должен делать. Правильная система обозначений создает организованную среду для реализации широкого круга удобных программных средств, а также создает костяк структуры для программ, которые пишут другие программы. Все мы хорошо владеем языками общего назначения, служащими для решения основной массы задач. Однако если задача столь специализирована и хорошо отработана, что ее программирование является почти механическим процессом, возможно, пора создать для нее систему обозначений, которая бы выражала ее естественным образом, а затем реализовать язык программирования для этой системы. Одним из наших любимых примеров являются регулярные выражения, однако есть и другие бессчетные возможности для создания специализированных мини-языков. Они совсем не должны быть сложными, чтобы давать отличные практические результаты.

Отдельные программисты нередко чувствуют себя маленькими винтиками в большой машине, поскольку им приходится использовать принудительно навязанные языки и системы, а также решать задачи, поставленные кем-то другим. Однако в дальней перспективе все же имеет значение лишь то, как хорошо мы выполняем свой долг. Применяя некоторые идеи, предложенные в этой книге, вы вскоре обнаружите, что с вашим кодом стало легче работать, отладка отнимает меньше времени и нервов, а стиль программирования стал более уверенным. Надеемся, что эта книга дала вам нечто для того, чтобы ваши усилия по программированию различных задач давали больше плодов и приносили больше удовлетворения.

Предметный указатель

Женщина: *Здесь ли моя тетушка Минни?*

Дрифтвуд: *Что ж, можете войти и поискать, если хотите; если даже ее здесь нет, всегда можно найти взамен кого-нибудь получше.*

Братья Маркс. “Ночь в опере”
(The Marx Brothers, *A Night at the Opera)*